Service Availability™ Forum Application Interface Specification

Availability Management Framework SAI-AIS-AMF-B.04.01

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

The Service Availability™ Forum Application Interface Specification (the "Package") found at the URL http://www.saforum.org is generally made available by the Service Availability Forum (the "Copyright Holder") for use in developing products that are compatible with the standards provided in the Specification. The terms and conditions which govern the use of the Package are covered by the Artistic License 2.0 of the Perl Foundation, which is reproduced here.

The Artistic License 2.0

Copyright (c) 2000-2006, The Perl Foundation.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed.

The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.

"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

Permission for Use and Modification Without Distribution

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

Permissions for Redistribution of the Standard Version

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

Distribution of Modified Versions of the Package as Source

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution.

If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

General Provisions

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty:

THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents — Availability Management Framework

1 Document Introduction 19

1.1 Document Purpose 19
1.2 AIS Documents Organization 19
1.3 History 19
1.3.1 New Topics 20
1.3.2 Clarifications 23
1.3.3 Deleted Topics 24
1.3.4 Other Changes 24
1.3.5 Superseded and Superseding Functions 28
1.3.6 Changes in Return Values of API and Administrative Functions 30
1.4 References 31
1.5 How to Provide Feedback on the Specification 31
1.6 How to Join the Service Availability™ Forum 32
1.7 Additional Information 32
1.7.1 Member Companies 32
1.7.2 Press Materials 32

2 Overview 33

2.1 Overview of the Availability Management Framework 33

3 System Description and System Model 35

3.1 Logical Entities 37
3.1.1 Cluster and Nodes 38
3.1.1.1 AMF Nodes 38
3.1.1.2 AMF Cluster 39
3.1.2 Components 41
3.1.2.1 SA-Aware Components 43
3.1.2.1.1 Container and Contained Components 44
3.1.2.2 Non-SA-Aware Components 46
3.1.2.2.1 External Components 46
3.1.2.2.2 Non-Proxied, Non-SA-Aware Components 46
3.1.2.2.3 Integration and Usage of Non-SA-Aware Local Components 47
3.1.2.3 Proxy and Proxied Components 48
3.1.2.4 Component Life Cycle 49
3.1.2.5 Component Type 50
3.1.3 Component Service Instance 50
3.1.3.1 Component Service Type 52
3.1.4 Service Unit 52
3.1.4.1 Service Unit Type 53
3.1.5 Service Instances 54
3.1.5.1 Service Type 55
3.1.6 Service Groups 55

3.1.6.1 Service Group Type 55
3.1.7 Application 56
3.1.7.1 Application Type 56
3.1.8 Protection Groups 56
3.1.9 Mapping of Service Units to Nodes 57
3.1.10 Service Unit Instantiation 58
3.1.11 Illustration of Logical Entities 59
3.2 State Models 61
3.2.1 Service Unit States 61
3.2.1.1 Presence State 61
3.2.1.2 Administrative State 63
3.2.1.3 Operational State 63
3.2.1.4 Readiness State 64
3.2.1.5 HA State of a Service Unit for a Service Instance 67
3.2.1.6 HA Readiness State of a Service Unit per Service Instance 69
3.2.2 Component States 71
3.2.2.1 Presence State 71
3.2.2.2 Operational State 75
3.2.2.3 Readiness State 76
3.2.2.4 HA State of a Component per Component Service Instance 77
3.2.2.5 HA Readiness State of a Component for a Component Service Instance 84
3.2.3 Service Instance States 87
3.2.3.1 Administrative State 87
3.2.3.2 Assignment State 88
3.2.4 Component Service Instance States 89
3.2.5 Service Group States 89
3.2.6 Node States 90
3.2.6.1 Administrative State 90
3.2.6.2 Operational State 91
3.2.7 Application States 92
3.2.8 Cluster States 93
3.2.9 Summary of States Supported for the Logical Entities 94
3.3 Fail-Over and Switch-Over 96
3.4 Possible Combinations of States for Service Units 98
3.4.1 Combined States for Pre-Instantiable Service Units 98
3.4.2 Combined States for Non-Pre-Instantiable Service Units 103
3.5 Component Capability Model 107
3.6 Service Group Redundancy Model 109
3.6.1 Common Characteristics 110
3.6.1.1 Common Definitions 110
3.6.1.2 Initiation of the Auto-Adjust Procedure for a Service Group 113
3.6.1.3 AMF Node Capacity Limitation 114
3.6.1.3.1 Examples 117
3.6.1.4 Considerations when Configuring Redundancy 120
3.6.2 2N Redundancy Model 122
3.6.2.1 Basics 122
3.6.2.2 Configuration 122

3.6.2.3 SI Assignments and Failure Handling . . . 123
3.6.2.3.1 Failure of the Active Service Unit . . . 123
3.6.2.3.2 Failure of the Standby Service Unit . . . 123
3.6.2.3.3 Auto-Adjust Procedure . . . 123
3.6.2.3.4 Cluster Startup . . . 124
3.6.2.3.5 Role of the List of Ordered Service Units in Assignments and Instantiations . . . 124
3.6.2.4 Examples . . . 125
3.6.2.5 UML Diagram of the 2N Redundancy Model . . . 131
3.6.3 N+M Redundancy Model . . . 132
3.6.3.1 Basics . . . 132
3.6.3.2 Examples . . . 133
3.6.3.3 Configuration . . . 135
3.6.3.4 SI Assignments . . . 137
3.6.3.4.1 Reduction Procedure . . . 139
3.6.3.5 Examples for Service Unit Fail-Over . . . 144
3.6.3.5.1 Handling of a Node Failure when Spare Service Units Exist . . . 144
3.6.3.5.2 Handling of a Node Failure when no Spare Service Units Exist . . . 145
3.6.3.6 Example of Auto-Adjust . . . 146
3.6.3.7 UML Diagram of the N+M Redundancy Model . . . 148
3.6.4 N-Way Redundancy Model . . . 149
3.6.4.1 Basics . . . 149
3.6.4.2 Example . . . 150
3.6.4.3 Configuration . . . 151
3.6.4.4 SI Assignments . . . 152
3.6.4.4.1 Reduction Procedure . . . 154
3.6.4.5 Failure Handling . . . 156
3.6.4.6 Example of Auto-Adjust . . . 158
3.6.4.7 UML Diagram of the N-Way Redundancy Model . . . 159
3.6.5 N-Way Active Redundancy Model . . . 160
3.6.5.1 Basics . . . 160
3.6.5.2 Example . . . 161
3.6.5.3 Configuration . . . 162
3.6.5.4 SI Assignments . . . 163
3.6.5.4.1 Reduction Procedure . . . 165
3.6.5.5 Failure Handling . . . 168
3.6.5.5.1 Example for Failure Recovery . . . 168
3.6.5.6 Example of Auto-Adjust . . . 172
3.6.5.7 UML Diagram of the N-Way Active Redundancy Model . . . 174
3.6.6 No-Redundancy Redundancy Model . . . 175
3.6.6.1 Basics . . . 175
3.6.6.2 Example . . . 176
3.6.6.3 Configuration . . . 177
3.6.6.4 SI Assignments . . . 177
3.6.6.4.1 Reduction Procedure . . . 178
3.6.6.5 Failure Handling . . . 179
3.6.6.6 Example of Auto-Adjust . . . 180
3.6.6.7 UML Diagram of the No-Redundancy Redundancy Model . . . 181
3.6.7 The Effect of Administrative Operations on Service Instance Assignments . . . 182
3.6.7.1 Locking a Service Unit or a Node . . . 182

3.6.7.2 Unlocking a Service Unit, a Service Group, or a Node 183
3.7 Component Capability Model and Service Group Redundancy Model 184
3.8 Dependencies Among SIs, Component Service Instances, and Components 185
3.8.1 Dependencies Among Service Instances and Component Service Instances 185
3.8.1.1 Dependencies Among SIs when Assigning a Service Unit Active for a Service Instance 185
3.8.1.2 Impact of Disabling a Service Instance on the Dependent Service Instances 186
3.8.1.3 Dependencies Among Component Service Instances of the same Service Instance 186
3.8.2 Dependencies Among Components 187
3.9 Approaches for Integrating Legacy Software or Hardware Entities 189
3.10 Component Monitoring 190
3.11 Error Detection, Recovery, Repair, and Escalation Policy 191
3.11.1 Basic Notions 191
3.11.1.1 Error Detection 191
3.11.1.2 Restart 191
3.11.1.3 Recovery 192
3.11.1.3.1 Restart Recovery Action 193
3.11.1.3.2 Fail-Over Recovery Action 194
3.11.1.3.3 Application Restart Recovery Action 197
3.11.1.3.4 Cluster Reset Recovery Action 197
3.11.1.4 Repair 197
3.11.1.4.1 Recovery and Associated Repair Policies 199
3.11.1.4.2 Restrictions to Auto-Repair 200
3.11.1.5 Recovery Escalation 201
3.11.2 Recovery Escalation Policy of the Availability Management Framework 201
3.11.2.1 Recommended Recovery Action 201
3.11.2.2 Escalations of Levels 1 and 2 202
3.11.2.3 Escalation of Level 3 205

4 Local Component Life Cycle Management Interfaces 207

4.1 Common Characteristics 207
4.2 Configuring the Pathname of CLC-CLI Commands 208
4.3 CLC-CLI Environment Variables 209
4.4 Configuring CLC-CLI Arguments 210
4.5 Exit Status 210
4.6 INSTANTIATE Command 211
4.7 TERMINATE Command 213
4.8 CLEANUP Command 214
4.9 AM_START Command 215
4.10 AM_STOP Command 215
4.11 Usage of CLC-CLI Commands Based on the Component Category 216

5 Proxied Components Management 217

5.1 Properties of Proxy and Proxied Components 217
5.2 Life Cycle Management of Proxied Components 218
5.3 Proxy Component Failure Handling 219

6 Contained Components Management 221
6.1 Overview of Container and Contained Components 221
6.1.1 Definitions 221
6.1.2 Component Category 221
6.1.3 Multiple Components per Process 221
6.1.4 Life Cycle Management of Contained Components 221
6.1.5 Container and Contained Components in Service Units and Service Groups 221
6.1.6 Redundancy Models 222
6.1.7 Administrative Operations and Container and Contained Components 223
6.1.8 Failure Handling 223
6.2 Life Cycle Management of Contained Components 224
6.2.1 Container CSI and Its Configuration 224
6.2.2 Assignment of the Container CSI 224
6.2.3 Life Cycle Callbacks 225
6.3 Failure Handling for Container and Contained Components 226
6.4 Proxied and Contained Components: Similarities and Differences 227

7 Availability Management Framework API 229
7.1 Availability Management Framework Model for the APIs 230
7.1.1 Callback Semantics and Component Registration and Unregistration 230
7.1.2 Component Healthcheck Monitoring 232
7.1.2.1 Overview 232
7.1.2.2 Variants of Healthchecks 233
7.1.2.3 Starting and Stopping Healthchecks 233
7.1.2.4 Healthcheck Configuration Issues 233
7.1.2.4.1 Role of Period and Maximum-Duration in Framework-Invoked Healthchecks 235
7.1.2.4.2 Role of Period in Component-Invoked Healthchecks 236
7.1.2.4.3 Modification of Healthcheck Parameters 237
7.1.3 Component Service Instance Management 237
7.1.4 Component Life Cycle Management 238
7.1.5 Protection Group Management 238
7.1.6 Error Reporting 238
7.1.7 Correlation of Notifications 239
7.1.8 Component Response to Framework Requests 239
7.1.9 API Usage Illustrations 240
7.2 Unavailability of the AMF API on a Non-Member Node 243
7.2.1 A Member Node Leaves or Rejoins the Cluster Membership 243
7.2.2 Guidelines for Availability Management Framework Implementers 244
7.3 Include File and Library Names 245
7.4 Type Definitions 245
7.4.1 SaAmfHandleT 245
7.4.2 Component Process Monitoring 245
7.4.2.1 SaAmfPmErrorsT Type 246
7.4.2.2 SaAmfPmStopQualifierT Type 246
7.4.3 Component Healthcheck Monitoring 246

7.4.3.1 SaAmfHealthcheckInvocationT 246
7.4.3.2 SaAmfHealthcheckKeyT 247
7.4.4 Types for State Management 247
7.4.4.1 HA State 247
7.4.4.2 Readiness State 247
7.4.4.3 Presence State 248
7.4.4.4 Operational State 248
7.4.4.5 Administrative State 248
7.4.4.6 Assignment State 249
7.4.4.7 HA Readiness State 249
7.4.4.8 Proxy Status 249
7.4.4.9 All Defined States 250
7.4.5 Component Service Types 250
7.4.5.1 SaAmfCSIFlagsT 250
7.4.5.2 SaAmfCSITransitionDescriptorT 251
7.4.5.3 SaAmfCSIStateDescriptorT 252
7.4.5.4 SaAmfCSIAttributeListT 253
7.4.5.5 SaAmfCSIDescriptorT 254
7.4.6 Types for Protection Group Management 255
7.4.6.1 SaAmfProtectionGroupMemberT_4 255
7.4.6.2 SaAmfProtectionGroupChangesT 255
7.4.6.3 SaAmfProtectionGroupNotificationT_4 256
7.4.6.4 SaAmfProtectionGroupNotificationBufferT_4 256
7.4.7 SaAmfRecommendedRecoveryT 257
7.4.8 SaAmfCompCategoryT 258
7.4.9 SaAmfRedundancyModelT 260
7.4.10 SaAmfCompCapabilityModelT 260
7.4.11 Notifications-Related Types 261
7.4.11.1 SaAmfNotificationMinorIdT 261
7.4.11.2 SaAmfAdditionalInfoIdT 262
7.4.12 SaAmfCallbacksT_4 263
7.5 Library Life Cycle 264
7.5.1 saAmfInitialize_4() 264
7.5.2 saAmfSelectionObjectGet() 267
7.5.3 saAmfDispatch() 268
7.5.4 saAmfFinalize() 270
7.6 Component Registration 272
7.6.1 saAmfComponentRegister() 272
7.6.2 saAmfComponentNameGet() 276
7.7 Passive Monitoring of Processes of a Component 278
7.7.1 saAmfPmStart_3() 278
7.7.2 saAmfPmStop() 280
7.8 Component Health Monitoring 283
7.8.1 saAmfHealthcheckStart() 283
7.8.2 SaAmfHealthcheckCallbackT 286
7.8.3 saAmfHealthcheckConfirm() 288
7.8.4 saAmfHealthcheckStop() 291

7.9 Component Service Instance Management . . . 293
7.9.1 saAmfHAStateGet() . . . 293
7.9.2 SaAmfCSISetCallbackT . . . 295
7.9.3 SaAmfCSIRemoveCallbackT . . . 298
7.9.4 saAmfCSIQuiescingComplete() . . . 300
7.9.5 saAmfHAReadinessStateSet() . . . 303
7.10 Component Life Cycle . . . 307
7.10.1 SaAmfComponentTerminateCallbackT . . . 307
7.10.2 SaAmfProxiedComponentInstantiateCallbackT . . . 309
7.10.3 SaAmfProxiedComponentCleanupCallbackT . . . 311
7.10.4 SaAmfContainedComponentInstantiateCallbackT . . . 312
7.10.5 SaAmfContainedComponentCleanupCallbackT . . . 314
7.11 Protection Group Management . . . 316
7.11.1 saAmfProtectionGroupTrack_4() . . . 316
7.11.2 SaAmfProtectionGroupTrackCallbackT_4 . . . 319
7.11.3 saAmfProtectionGroupTrackStop() . . . 322
7.11.4 saAmfProtectionGroupNotificationFree_4() . . . 323
7.12 Error Reporting . . . 325
7.12.1 saAmfComponentErrorReport_4() . . . 325
7.12.2 saAmfComponentErrorClear_4() . . . 327
7.12.3 saAmfCorrelationIdsGet() . . . 330
7.13 Component Response to Framework Requests . . . 333
7.13.1 saAmfResponse_4() . . . 333

8 AMF UML Information Model . . . 337

8.1 Use of Entity Types in the AMF UML Information Model . . . 338
8.2 Notes on the Conventions Used in UML Diagrams . . . 338
8.3 DN Formats for Availability Management Framework UML Classes . . . 339
8.4 AMF Cluster . . . 341
8.5 Availability Management Framework Instances and Types View . . . 342
8.6 Availability Management Framework Instances View . . . 343
8.7 AMF Cluster, Node, and Node-Related Classes . . . 344
8.8 Application Classes Diagram . . . 346
8.9 Service Group Class Diagram . . . 348
8.10 Service Unit Class Diagram . . . 350
8.11 Service Instance Class Diagram . . . 353
8.12 Component Service Instance Diagram . . . 356
8.13 Component and Component Types Class Diagrams . . . 358
8.13.1 Component Type Class Diagram . . . 358
8.13.2 Component Classes Diagram . . . 360
8.14 AMF Global Component Attributes and Healthcheck Classes . . . 362

9 Administration API . . . 365

9.1 Availability Management Framework Administration API Model . . . 365
9.2 Include File and Library Name . . . 367
9.3 Type Definitions . . . 367
9.3.1 SaAmfAdminOperationIdT . . . 367
9.4 Availability Management Framework Administration API . . . 368
9.4.1 Administrative State Modification Operations . . . 368
9.4.2 SA_AMF_ADMIN_UNLOCK . . . 370
9.4.3 SA_AMF_ADMIN_LOCK . . . 372
9.4.4 SA_AMF_ADMIN_LOCK_INSTANTIATION . . . 375
9.4.5 SA_AMF_ADMIN_UNLOCK_INSTANTIATION . . . 378
9.4.6 SA_AMF_ADMIN_SHUTDOWN . . . 380
9.4.7 SA_AMF_ADMIN_RESTART . . . 383
9.4.8 SA_AMF_ADMIN_SI_SWAP . . . 386
9.4.9 SA_AMF_ADMIN_SG_ADJUST . . . 389
9.4.10 SA_AMF_ADMIN_REPAIRED . . . 391
9.4.11 SA_AMF_ADMIN_EAM_START . . . 393
9.4.12 SA_AMF_ADMIN_EAM_STOP . . . 395
9.5 Summary of Administrative Operation Support . . . 397

10 Basic Operational Scenarios . . . 399

10.1 Administrative Shutdown of a Service Instance in a 2N Case . . . 399
10.2 Administrative Shutdown of a Service Unit in a 2N Case . . . 401
10.3 Administrative Shutdown of a Service Unit for the N-Way Model . . . 402
10.4 Administrative Lock of a Service Instance . . . 404
10.5 Administrative Lock of a Service Unit . . . 405
10.6 A Simple Fail-Over . . . 406
10.7 Administrative Shutdown of an SI Having a Container CSI . . . 407
10.8 Administrative Lock of an SI Having a Container CSI . . . 410
10.9 Administrative Lock of a Service Unit with a Container Component . . . 411
10.10 Restart of a Container Component . . . 414

11 Alarms and Notifications . . . 417

11.1 Setting Common Attributes . . . 417
11.2 Availability Management Framework Notifications . . . 419
11.2.1 Availability Management Framework Alarms . . . 419
11.2.1.1 Component Instantiation Failed . . . 419
11.2.1.2 Component Cleanup Failed . . . 421
11.2.1.3 Cluster Reset Triggered by a Component Failure . . . 423
11.2.1.4 Service Instance Unassigned . . . 425
11.2.1.5 Proxy Status of a Component Changed to Unproxied . . . 427
11.2.2 Availability Management Framework State Change Notifications . . . 428
11.2.2.1 Administrative State Change Notify . . . 428
11.2.2.2 Operational State Change Notify . . . 429
11.2.2.3 Presence State Change Notify . . . 430

11.2.2.4 HA State Change Notify 431
11.2.2.5 HA Readiness State Change Notify 432
11.2.2.6 SI Assignment State Change Notify 433
11.2.2.7 Proxy Status of a Component Changed to Proxied 434
11.2.3 Availability Management Framework Notifications of Miscellaneous Type 435
11.2.3.1 Error Report Notification 435
11.2.3.2 Error Clear Notification 437

Appendix A Implementation of CLC Interfaces 439

Appendix B API Functions and Registered Processes 441

Appendix C Example for Proxy/Proxied Association 443

Appendix D Interaction with CLM 445

Index of Definitions 447

List of Figures

Figure 1: Availability Management Framework Logical Entities and Their Relations . . . 37
Figure 2: Elements of the System Model . . . 60
Figure 3: State Diagram of the HA State of an SA-Aware Component for a CSI . . . 83
Figure 4: State Transitions for Pre-Instantiable Service Units . . . 102
Figure 5: State Transitions for Non-Pre-Instantiable Service Units . . . 106
Figure 6: Example 1 for Node Capacity Limitation . . . 118
Figure 7: Example 2 for Node Capacity Limitation . . . 119
Figure 8: Example of the 2N Redundancy Model: Two Service Units on Different Nodes . . . 125
Figure 9: Example of the 2N RM. Two SUs on Different Nodes, Fault Has Occurred . . . 126
Figure 10: Example of the 2N Redundancy Model: Two Service Units on the Same Node . . . 127
Figure 11: Example of the 2N RM: Two SUs on the Same Node, Fault Has Occurred . . . 128
Figure 12: Example of the 2N RM: One Node Provides Standby SUs for Several Service Groups . . . 129
Figure 13: Example of the 2N RM: Each Node Has an Active and a Standby Service Unit . . . 130
Figure 14: UML Diagram for the 2N Redundancy Model . . . 131
Figure 15: Example of the N+1 Redundancy Model . . . 134
Figure 16: Example of the N+M Redundancy Model, Where N = 3 and M = 2 . . . 135
Figure 17: UML Diagram of the N+M Redundancy Model . . . 148
Figure 18: Example of the N-Way Redundancy Model . . . 150
Figure 19: UML Diagram of the N-Way Redundancy Model . . . 159
Figure 20: Example of the N-Way Active Redundancy Model . . . 161
Figure 21: UML Diagram of the N-Way Active Redundancy Model . . . 174
Figure 22: Example of the No-Redundancy Redundancy Model . . . 176
Figure 23: UML Diagram of the No-Redundancy Redundancy Model . . . 181
Figure 24: SA-Aware Component Consisting of a Single Process . . . 240
Figure 25: SA-Aware Component Consisting of Multiple Processes . . . 241
Figure 26: A Single-Process Proxy Component and Two Proxied Components . . . 242
Figure 27: 3- Cluster View . . . 341
Figure 28: 3.1- AMF Instances and Types View . . . 342
Figure 29: 3.2- AMF Instances View . . . 343
Figure 30: 3.3- AMF Cluster, Node, and Node-Related Classes . . . 345
Figure 31: 3.4- AMF Application Classes . . . 347
Figure 32: 3.5- AMF SG Classes . . . 349
Figure 33: 3.6- AMF SU Classes . . . 352
Figure 34: 3.7- AMF SI Classes . . . 355
Figure 35: 3.8- AMF CSI Classes . . . 357
Figure 36: 3.9b- AMF Component Type Classes . . . 359
Figure 37: 3.9a- AMF Component Classes . . . 361
Figure 38: 3.9c- AMF Global Component Attributes and Healthcheck Classes . . . 363
Figure 39: Administrative States and Related Operations for AMF Entities . . . 369
Figure 40: Administrative Shutdown of a Service Instance for the 2N Case . . . 400
Figure 41: Administrative Shutdown of a Service Unit for the 2N Case . . . 401

Figure 42: Administrative Shutdown of a Service Unit for the N-Way Case 403
Figure 43: Administrative Lock of a Service Instance for the 2N Case 404
Figure 44: Administrative Lock of a Service Unit for the 2N Case 405
Figure 45: Fail-Over Scenario for a Service Group with the 2N Redundancy Model 406
Figure 46: Scenario for Shutting Down a Service Instance Having a Container CSI 408
Figure 47: Administrative Shutdown of a Service Instance Having a Container CSI 409
Figure 48: Administrative Lock of a Service Instance Having a Container CSI 410
Figure 49: Scenario for Locking a Service Unit Containing a Container Component 412
Figure 50: Administrative Lock of a Service Unit Containing a Container Component 413
Figure 51: Restart of a Container Component 415

List of Tables

Table 1: Superseded Functions and Type Definitions in Version B.04.01 . . . 29
Table 2: Changes in Return Values of API and Administrative Functions . . . 30
Table 3: Component Categories . . . 50
Table 4: Service Unit's Readiness State . . . 67
Table 5: Presence State of Components of a Service Unit . . . 74
Table 6: Component's Readiness State . . . 77
Table 7: HA State of Component/Component Service Instance . . . 79
Table 8: Application Developer View for Pre-Instantiable Components . . . 84
Table 9: Application Developer View for Non-Pre-Instantiable Components . . . 84
Table 10: Combinations of States for a Component . . . 87
Table 11: Preferred Number of Active and Standby Assignments . . . 88
Table 12: Summary of States Supported for the Logical Entities . . . 94
Table 13: Combined Administrative States for the Service Unit . . . 98
Table 14: Combined States for Pre-Instantiable Service Units . . . 99
Table 15: Combined States for Non-Pre-Instantiable Service Units . . . 104
Table 16: Component Capability Model and Service Group Redundancy Model . . . 184
Table 17: Recovery and Associated Automatic Repair Policies . . . 200
Table 18: Levels of Escalation . . . 202
Table 19: Usage of CLC-CLI Commands Based on the Component Category . . . 216
Table 20: Possible Combinations of Values in SaAmfCompCategoryT . . . 259
Table 21: DN Formats . . . 339
Table 22: Summary: Applicability of Administrative Operations . . . 397
Table 23: Component Instantiation Failed Alarm . . . 420
Table 24: Component Cleanup Failed Alarm . . . 422
Table 25: Cluster Reset Triggered by a Component Failure Alarm . . . 424
Table 26: Service Instance Unassigned Alarm . . . 426
Table 27: Proxy Status of a Component Changed to Unproxied Alarm . . . 427
Table 28: Administrative State Change Notification . . . 428
Table 29: Operational State Change Notification . . . 429
Table 30: Presence State Change Notification . . . 430
Table 31: HA State Change Notification . . . 431
Table 32: HA Readiness State Change Notification . . . 432
Table 33: SI Assignment State Change Notification . . . 433
Table 34: Proxy Status of a Component Changed to Proxied Notification . . . 434
Table 35: Error Report Notification . . . 436
Table 36: Error Clear Notification . . . 437
Table 37: Implementation of CLC Operations for Each Component Category . . . 439
Table 38: API Functions and Registered Processes . . . 441

1 Document Introduction

1.1 Document Purpose

This document defines the Availability Management Framework of the Application Interface Specification (AIS) of the Service Availability™ Forum (SA Forum). It is intended for use by implementers of the Application Interface Specification and by application developers who would use the Application Interface Specification to develop applications that must be highly available. The AIS is defined in the C programming language and requires substantial knowledge of the C programming language.

Typically, the Service Availability™ Forum Application Interface Specification will be used in conjunction with the Service Availability™ Forum Hardware Platform Interface Specification (HPI).

1.2 AIS Documents Organization

The Application Interface Specification is organized into several volumes. For a list of all Application Interface Specification documents, refer to the SA Forum Overview document [1].

1.3 History

Previous releases of the Availability Management Framework specification:

(1) SAI-AIS-AMF-A.01.01
(2) SAI-AIS-AMF-B.01.01
(3) SAI-AIS-AMF-B.02.01
(4) SAI-AIS-AMF-B.03.01

This section presents the changes of the current release, SAI-AIS-AMF-B.04.01, with respect to the SAI-AIS-AMF-B.03.01 release. Editorial changes that do not change semantics or syntax of the described interfaces are not mentioned.

1.3.1 New Topics

⇒ The HA readiness state of a service unit for a service instance and the HA readiness state of a component for a component service instance have been introduced. The main changes to the specification due to this topic are:

- The HA readiness state of a service unit for a service instance and the HA readiness state of a component for a component service instance have been introduced in Section 3.2.1.6 and Section 3.2.2.5, respectively. Due to the latter change, Section 3.2.2.4 has been adapted to explain that a component can reject a CSI assignment. Table 12 in Section 3.2.9 has also been updated accordingly.
- The `SaAmfHaReadinessStateT` type has been introduced in Section 7.4.4.7, and this extension induced a change to Section 7.4.4.9.
- As the HA readiness state of a component for a component service instance is reflected in the protection group track functions, the structures defined in Section 7.4.6.1, Section 7.4.6.2 (only the description), Section 7.4.6.3, and Section 7.4.6.4 have also been affected. As a consequence, superseding structures `SaAmfProtectionGroupMemberT_4` (Section 7.4.6.1), `SaAmfProtectionGroupNotificationT_4` (Section 7.4.6.3), and `SaAmfProtectionGroupNotificationBufferT_4` (Section 7.4.6.4) have been introduced.
  As a consequence of the previous changes, superseding functions `SaAmfProtectionGroupTrack_4()`, `SaAmfProtectionGroupTrackCallbackT_4`, and `saAmfProtectionGroupNotificationFree_4()`, which are described in Section 7.11.1, Section 7.11.2, and Section 7.11.4, respectively, have also been introduced.
- The `SaAmfCSISetCallbackT` function (see Section 7.9.2) has been extended, so that the component can return the new value `SA_AIS_ERR_NOT_READY` in the `error` parameter when replying to the callback to indicate that the component cannot assume the HA state specified by `haState` for the given component service instance.
- The sentence “`SA_AMF_TARGET_ALL` is always set for components that support only the “x_active_or_y_standby” capability model” has been removed from the description of the `SaAmfCSIRemoveCallbackT` function of the B.03.01 version (this section is now Section 7.9.2).
- The `saAmfHAReadinessStateSet()` function (see Section 7.9.5) has been introduced to enable a component to set its HA readiness state for component service instances.

- Two minor changes have been made to the `saAmfProtectionGroupTrack_4()` function (see Section 7.11.1) to track changes of the HA readiness state.
- The runtime attributes `saAmfSISUHAReadinessState` and `saAmfCSICompHAReadinessState` have been added to the object classes `SaAmfSIAssignment` (see FIGURE 34 in Section 8.11) and `SaAmfCSIAssignment` (see FIGURE 35 in Section 8.12), respectively. The definitions of these two object classes have been updated accordingly.
- A new notification has been introduced; it is issued when the HA readiness state of a service unit for an assigned service instance changes. Refer to the new Section 11.2.2.5.

⇒ Active or standby assignments of service instances to service units hosted by an AMF node impose a certain load on the AMF node in terms of resources like memory or computing power. In certain cases, AMF nodes having limited capacity for these resources can be overloaded due to these assignments. The Availability Management Framework specification has been extended to provide configuration attributes to help the Availability Management Framework avoiding to overload an AMF node with more service instance assignments that the AMF node can handle. The main changes induced by this topic to the specification are:

- The description of the “ordered list of SIs” in Section 3.6.1.1 has been changed to state that the rank of an SI is now global to the cluster.
- Section 3.6.1.3 has been introduced; it contains the main changes and provides examples.
- Section 3.6.1.4 has been introduced to provide some guidelines for a system architect to configure redundancy by using the different options provided by the Availability Management Framework.
- The `saAmfNodeCapacity` configuration attribute has been added to the `SaAmfNode` object class (see FIGURE 30 in Section 8.7).
- The configuration attributes `saAmfSIActiveWeight` and `saAmfSIStandbyWeight` have been added to the `SaAmfSI` object class (see FIGURE 34 in Section 8.11). Default values for the latter two attributes have been defined by the `saAmfSvcDefActiveWeight` and `saAmfSvcDefStandbyWeight` attributes in the `SaAmfSvcType` object class, shown also in FIGURE 34.

⇒ To support correlation Ids, the following main changes have been made:

- An overview of the Availability Management Framework's handling of correlation ids is presented in the new Section 7.1.7 on page 239.
- The `SA_AMF_AI_RECOMMENDED_RECOVERY` and `SA_AMF_AI_APPLIED_RECOVERY` values have been added to the `SaAmfAdditionalInfoIdT` enum (see Section 7.4.11.2 on page 262).
- The `correlationIds` parameter has been added to the superseding functions `saAmfComponentErrorReport_4()`, `saAmfComponentErrorClear_4()`, and `saAmfResponse_4()`, which are described in Section 7.12.1 on page 325, Section 7.12.2 on page 327, and on Section 7.13.1 on page 333, respectively.
- The `saAmfCorrelationIdsGet()` function has been defined (see Section 7.12.3 on page 330).
- New notifications have been specified. Refer to Section 11.2.3 on page 435.

⇒ To align the Availability Management Framework specification with the Platform Management Service (abbreviated as PLM, see [5]), the following modifications have been made:

- The notion of physical node has been replaced by the notion of PLM execution environment. This replacement induced adaptations in the definition of AMF node and AMF Cluster, see Section 3.1.1.1 on page 38 and Section 3.1.1.2 on page 39, respectively. Additionally, the definitions of local and external resources as well as the definitions of local and external components have been adapted (see Section 3.1.2).
- The description of the "node failfast" recovery action in Section 3.11.1.3.2 has been adapted.
- The interactions between the Availability Management Framework and the Cluster Membership Service have been described in the new Appendix D on page 445, which also contains references to PLM.

⇒ The `SaAmfNotificationMinorIdT` type (see Section 7.4.11.1 on page 261) has been introduced to define the `minorId` field of notifications produced by the Availability Management Framework. The descriptions of the notifications in Section 11.2 refer to this type.

1.3.2 Clarifications

⇒ In Section 3.2.1.4, it is clarified that pre-instantiable service units may be instantiated while they are out-of-service.

⇒ Section 3.2.2.2 adds one more case to the list of component failures detected by the Availability Management Framework, namely when `saAmfProxiedComponentInstantiateCallback()` or `saAmfContainedComponentInstantiateCallback()` function returns with an error.

⇒ Text has been added to Section 3.2.9 to clarify the interdependency between the states of components and the states of their containing service unit.

⇒ The role of the ordered list of service units in assignments and instantiations has been clarified (see Section 3.6.2.3.5).

⇒ The description of repair in the B.03.01 version of the Availability Management Framework concentrated mainly on the `SA_AMF_ADMIN_REPAIRED` administrative operation, and the usage of the `saAmfComponentErrorClear()` function as a possible way to perform a repair action was not always mentioned. The B.04.01 version of the specification extends the description of repair to appropriately refer to the `saAmfComponentErrorClear_4()` function as follows:

- References to the `saAmfComponentErrorClear_4()` function have been added to Section 3.4.1 and to Section 3.11.1.4 (repair).
- References to the extended Section 3.11.1.4 have been added to Section 4.6 (`INSTANTIATE` command) and Section 4.8 (`CLEANUP` command).
- The description section of the `saAmfComponentErrorClear_4()` function in Section 7.12.2 has been extended.

⇒ In Section 6.3, it is clarified that if a contained component fails, it is the task of the container component to report an error on the failed component.

⇒ This version of the Availability Management Framework specification clarifies that the registered process for a proxied component may differ from the registered process for the proxy component. This clarification affects Section 7.1.1, Section 7.6.1, Section 7.10.2, and Section 7.10.3.

⇒ The interpretation of the `SA_AMF_CSI_STILL_ACTIVE` value in Section 7.4.5.2 has been clarified.

⇒ In the description of the `saAmfInitialize_4()` function in Section 7.5.1 on page 264, it is clarified that the handle `amfHandle` is finalized when the Availability Management Framework detects the death of the invoking process.

⇒ Section 9.4.2 up to Section 9.4.6 specify the administrative states that the logical entities must have as a precondition to apply the corresponding administrative operations.

1.3.3 Deleted Topics

⇒ The `saAmfComponentUnregister()` function has been removed, as the unregistration of a component is now implicitly done by the Availability Management Framework. The term “unregistered process” has also been removed. These deletions implied several changes to the document. In particular, it is stated now that if a proxied component fails, it is the task of its proxy to report an error on the failed component.

⇒ According to SA Forum directives, AIS Services and Frameworks shall only generate alarms for situations that require an explicit intervention by an external agent or operator, provided that the corrective measures to be taken are well defined. Based on these directives, it was decided to remove the "service impaired” alarm from the Availability Management Framework B.04.01 version. SA Forum does not mandate that Availability Management Framework implementations which also support the B.03.01 version must generate the "service impaired” alarm for the B.03.01 version.

1.3.4 Other Changes

⇒ The definition of the term regular SA-aware component was changed to apply only to those components that only have the SA_AWARE flag set, that is, a regular SA-aware component cannot be a proxy component. For this purpose, the definition of this term in Section 3.1.2.1 was changed. Additionally, Table 3 and Table 20 were adapted, and the term regular SA-aware component was used accordingly in Chapter 10.

⇒ In Section 3.1.2.1.1, it is stated that a process of a contained component must also belong to its associated container component.

⇒ The definition of instantiable service unit has been extended, and the description of the reduction procedure has been clarified. See Section 3.6.1.1.

⇒ In Section 3.6.6.1 on the no-redundancy redundancy model, it is explained that the Availability Management Framework can recover from faults by failing over the active assignment to a spare service unit.

⇒ As the `saAmfSGNumPrefAssignedSUs` attribute is not defined for the no-redundancy redundancy model, the sentence “Note that the preferred number of assigned service units is equal to the number of configured SIs plus one spare service unit." on the configuration of this redundancy model has been removed (the corresponding section in this version is Section 3.6.6.3).

⇒ In Section 7.2.1, it is explained that when the Availability Management Framework detects that a CLM node has unexpectedly left the cluster, the Availability Management Framework abruptly terminates the components hosted by the AMF node that is mapped to this CLM node. This modification led also to a change in the explanation contained in Appendix A regarding when to invoke the `saAmfProxiedComponentCleanupCallback()` callback or to execute the `CLEANUP` command.

⇒ The one by one assignment of component service instances to a component and the one by one removal of such assignments from a component with an x_active_or_y_standby capability model are now allowed. Modifications have been made to the description of the `SA_AMF_CSI_TARGET_ALL` flag in Section 7.4.5.1 and to the description of the `SaAmfCSIRemoveCallbackT` function in Section 7.9.3.

⇒ In Section 7.4.8, a typo in the name of the `saAmfCompCategoryT` type definition has been corrected, as it should be `SaAmfCompCategoryT`. Similar changes were made for `SaAmfRedundancyModelT` in Section 7.4.9 and for `SaAmfCompCapabilityModelT` in Section 7.4.10. Additonally, in Section 7.4.9, the `SA_AMF_N-WAY_REDUNDANCY-MODEL` name has been corrected to `SA_AMF_N_WAY_REDUNDANCY_MODEL`.

⇒ Also in Section 7.4.8, a value for `SA_AMF_COMP_PROXIED_NPI` was introduced as a category value for proxied, non-pre-instantiable components. This addition implied modifications in Table 20 in the same section. A clarification was also included to explain the possible category values for pre-instantiable and non-pre-instantiable components.

  A further clarification regarding pre-instantiable and non-pre-instantiable components has been added to Section 8.13.1 on the `SaAmfCtCsType` association class.

⇒ The sentence “On return from the `saAmfResponse()` function, the Availability Management Framework removes all service instances associated with the component and the component terminates.” was removed from the description of the `SaAmfComponentTerminateCallbackT` function (see Section 7.10.1 in this document), as this sentence is not always true.

⇒ Version B.03.01 of this specification stated in the description of the `SaAmfProxiedComponentInstantiateCallbackT` callback function that the invoked proxy component must have previously registered the proxied component with the Availability Management Framework. This statement is not true; the correct sequence of operations was already described in the example for pre-instantiable components in Appendix C.

The pertinent correction applies to the `SaAmfProxiedComponentInstantiateCallbackT` function (see Section 7.10.2) and to the `SaAmfCSISetCallbackT` function (see Section 7.9.2). The description of the latter callback function clarifies that the invoked process must register a non-pre-instantiable proxied component before it responds to this callback request.

In the process of making these changes, the descriptions of several functions involving proxied or contained components have been extended to specify for which errors and for which component (proxy or proxied, container or contained) the recovery policy applies.

- `saAmfHealthcheckStart()` (see Section 7.8.1).
- `SaAmfHealthcheckCallbackT` (see Section 7.8.2).
- `saAmfHealthcheckConfirm()` (see Section 7.8.3).
- `SaAmfCSISetCallbackT` (see Section 7.9.2).
- `SaAmfCSIRemovedCallbackT` (see Section 7.9.3).
- `SaAmfComponentTerminateCallbackT` (see Section 7.10.1).
- `SaAmfProxiedComponentInstantiateCallbackT` (see Section 7.10.2).
- `SaAmfProxiedComponentCleanupCallbackT` (see Section 7.10.3).
- `SaAmfContainedComponentInstantiateCallbackT` (see Section 7.10.4).
- `SaAmfContainedComponentCleanupCallbackT` (see Section 7.10.5).

This correction induced also the following modifications:

- The descriptions of several functions called by a component have been extended such that the sentences also apply when a proxy or container component performs the particular actions on itself and on the proxied or contained components, respectively. Similar changes were made also for callback functions.
- Textual modifications have also been made in the description of several functions to use the text “the registered process” instead of “a registered process”, as at most one registered process exists for a component.
- The last paragraph in Section 4.8 on the `CLEANUP` command has been extended to clarify that the explanation also applies to all contained components if the affected component is a container component.
- A paragraph in Section 5.2 has been extended to clarify the type of the affected proxied component (pre-instantiable or non-pre-instantiable).

- A paragraph in Section 5.3 clarifies that an already instantiated proxied component need not be reinstantiated if the a new proxy takes over the task of controlling the proxied component.
- The description of the `saAmfComponentRegister()` function in Section 7.6.1 specifies additional cases when this function is called.
- Section 7.8.1 on the `saAmfHealthcheckStart()` function clarifies to which component the invoking process must belong and when the Availability Management Framework automatically stops a healthcheck. Additionally, this section clarifies the conditions to start a healthcheck for a proxied component.
- Section 7.8.2 on the `SaAmfHealthcheckCallbackT` function clarifies that this callback is called on the same process that started the healthcheck operation by invoking the `saAmfHealthcheckStart()` function.
- Section 7.8.3 on the `saAmfHealthcheckConfirm()` function clarifies that the invoking process must be the same process that started the healthcheck operation by invoking the `saAmfHealthcheckStart()` function.
- Section 7.9.4 on the `saAmfCSIQuiescingComplete()` function clarifies that the component that was requested to quiesce was the component identified by the name referred to by the `compName` parameter in the corresponding invocation of the `saAmfCSISetCallback()` callback function. If the `error` parameter is set to `SA_AIS_ERR_FAILED_OPERATION`, the Availability Management Framework must engage the configured recovery policy for the component referred to by the `compName` parameter in the corresponding `saAmfCSISetCallback()` callback function.

⇒ The following typos have been corrected in Table 21 on DN formats:

- Object class `SaAmfSGType`: `safSgType` instead of `SafSgType`.
- Object class `SaAmfSU`: `safSg` instead of `safSG`.
- Object class `SaAmfSutCompType`: `safSuType` instead of `SafSuType`.
- Object class `SaAmfSUType`: `safSuType` instead of `SafSuType`.

⇒ The DN format for the `SaAmfCSType` object class in Table 21 has been corrected.

⇒ The configuration attributes `saAmfAppType`, `saAmfSGType`, `saAmfSUType`, `saAmfSvcType`, `saAmfCSType`, and `saAmfCompType`, shown in the UML diagrams in Chapter 8, have been made writable.

⇒ In FIGURE 28 in Section 8.5, to align with other similar associations like the one between the SaAmfSUType and SaAmfCompType classes, the shared aggregations between the SaAmfAppType and SaAmfSGType classes and between the SaAmfSGType and SaAmfSUType classes have been changed to associations that are navigable only in one direction.
Additionally, the multiplicity between the SaAmfHealthcheck and SaAmfHealthcheckType classes (at the latter end) has been changed to 1.

⇒ In the SaAmfComp class, which is described in Section 8.13.2 on page 360, the following changes have been made:

- to correct typos in the preceding version, the term 'Comp' has been dropped from the default values of the following attributes, as they refer to the appropriate attributes of the SaAmfCompGlobalAttributes class, described in Section 8.14 on page 362:
  - saAmfCompNumMaxInstantiateWithoutDelay,
  - saAmfCompNumMaxInstantiateWithDelay,
  - saAmfCompNumMaxAmStartAttempts, and
  - saAmfCompNumMaxAmStopAttempts;
- to align the terminate operation with the cleanup and instantiate operations, which have just one timeout attribute, regardless of whether the operation is invoked by a CLC-CLI or by a callback call, the saAmfCompTerminateCallbackTimeout attribute has been dropped. The saAmfCompTerminateTimeout attribute applies now to both ways of invoking the operation, and the name of its default value has been changed to saAmfCtDefCallbackTimeout to reflect that components are typically terminated by invoking the callback call. Note also that a typo in this latter name ("TimeOut" instead of "Timeout") has been corrected in the component type class diagram in Section 8.13.1 on page 358.

⇒ In Section 9.3.1, a typo in the name of the saAmfAdminOperationIdT type definition has been corrected, as it should be SaAmfAdminOperationIdT.

⇒ In Section 9.4.6 and Section 9.5, a typo has been corrected: operationId is SA_AMF_ADMIN_SHUTDOWN and not SA_AMF_ADMIN_SHUT_DOWN.

⇒ The SaAmfProtectionGroupNotificationFree_4() function has been included in Table 38 in Appendix B.

1.3.5 Superseded and Superseding Functions

The Availability Management Framework defines for the version B.04.01 new functions and new type definitions to replace functions and type definitions of the version B.03.01. The list of replaced functions and type definitions in alphabetic order is presented in Table 1.

The superseded functions and type definitions are no longer supported in version B.04.01, and no description is provided for them in this document. The names of the superseding functions and type definitions are obtained by adding "_4" to the respective names of the B.03.01 version or by replacing "_*n*" (where *n* is a number < =3) by "_4" if the superseded functions or type definitions had already "_*n*" at the end of their names. Regarding the support of backward compatibility in SA Forum AIS, refer to [2].

Table 1 Superseded Functions and Type Definitions in Version B.04.01

Functions and Type Definitions of Version B.03.01 no Longer Supported in B.04.01
`SaAmfCallbacksT_3`
`saAmfComponentErrorClear()`
`saAmfComponentErrorReport()`
`saAmfComponentUnregister`[1]`()`
`saAmfInitialize_3()`
`SaAmfProtectionGroupMemberT`
`SaAmfProtectionGroupNotificationBufferT`
`SaAmfProtectionGroupNotificationFree()`
`SaAmfProtectionGroupNotificationT`
`SaAmfProtectionGroupTrack()`
`SaAmfProtectionGroupTrackCallbackT`
`SaAmfResponse()`

1. As an exception, this function has not been superseded; instead, it has been only removed, as the unregistration is implicitly done by the Availability Management Framework.

1.3.6 Changes in Return Values of API and Administrative Functions

The following table applies only to functions that have not been superseded.

Table 2 Changes in Return Values of API and Administrative Functions

Function	Return Value	Change Type
All administrative operations described in Chapter 9	SA_AIS_ERR_TIMEOUT SA_AIS_ERR_NO_MEMORY	new
SA_AMF_ADMIN_SI_SWAP administrative operation	SA_AIS_ERR_BAD_OPERATION	extended
saAmfComponentRegister()	SA_AIS_ERR_INIT	changed
SaAmfContainedComponentInstantiateCallbackT[1]	SA_AIS_ERR_TRY_AGAIN SA_AIS_OK	extended
saAmfCSIQuiescingComplete()	SA_AIS_ERR_NOT_EXIST	new
SaAmfCSISetCallbackT[1]	SA_AIS_ERR_NOT_READY	new
saAmfHealthcheckStart()	SA_AIS_ERR_NOT_EXIST	extended
saAmfPmStart_3()[2]	SA_AIS_ERR_VERSION	new
SaAmfProxiedComponentInstantiateCallbackT[1]	SA_AIS_ERR_TRY_AGAIN	extended

1. Actually, this callback function does not return any value. The value shown in the second column is the value set in the `error` parameter when the invoked process responds to the callback function by invoking the `saAmfResponse_4()` function.
2. This return value should have been added in the Availability Management Framework B.03.01 specification to this function.

1.4 References

The following documents contain information that is relevant to specification:

[1] Service Availability™ Forum, Service Availability Interface, Overview, SAI-Overview-B.05.01

[2] Service Availability™ Forum, Service Availability Interface, C Programming Model, SAI-AIS-CPROG-B.05.01

[3] Service Availability™ Forum, Application Interface Specification, Notification Service, SAI-AIS-NTF-A.03.01

[4] Service Availability™ Forum, Application Interface Specification, Cluster Membership Service, SAI-AIS-CLM-B.04.01

[5] Service Availability™ Forum, Application Interface Specification, Platform Management Service, SAI-AIS-PLM-A.01.01

[6] Service Availability™ Forum, Application Interface Specification, Information Model Management Service, SAI-AIS-IMM-A.03.01

[7] Service Availability™ Forum, Information Model in XML Metadata Interchange (XMI) v2.1 format, SAI-IM-XMI-A.04.01.xml.zip

[8] Service Availability™ Forum, Application Interface Specification, Software Management Service, SAI-AIS-SMF-A.01.01

[9] CCITT Recommendation X.731 | ISO/IEC 10164-2, State Management Function

[10] CCITT Recommendation X.733 | ISO/IEC 10164-4, Alarm Reporting Function

[11] IETF RFC 2253 (http://www.ietf.org/rfc/rfc2253.txt).

[12] IETF RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

References to these documents are made by putting the number of the document in brackets.

1.5 How to Provide Feedback on the Specification

If you have a question or comment about this specification, you may submit feedback online by following the links provided for this purpose on the Service Availability™ Forum Web site (http://www.saforum.org).

You can also sign up to receive information updates on the Forum or the Specification.

1.6 How to Join the Service Availability™ Forum

The Promoter Members of the Forum require that all organizations wishing to participate in the Forum complete a membership application. Once completed, a representative of the Service Availability™ Forum will contact you to discuss your membership in the Forum. The Service Availability™ Forum Membership Application can be completed online by following the pertinent links provided on the SA Forum Web site (http://www.saforum.org).

You can also submit information requests online. Information requests are generally responded to within three business days.

1.7 Additional Information

1.7.1 Member Companies

A list of the Service Availability™ Forum member companies can be viewed online by using the links provided on the SA Forum Web site (http://www.saforum.org).

1.7.2 Press Materials

The Service Availability™ Forum has available a variety of downloadable resource materials, including the Forum Press Kit, graphics, and press contact information. Visit this area often for the latest press releases from the Service Availability™ Forum and its member companies by following the pertinent links provided on the SA Forum Web site (http://www.saforum.org).

2 Overview

This specification defines the Availability Management Framework within the Application Interface Specification (AIS).

2.1 Overview of the Availability Management Framework

The Availability Management Framework (sometimes also called the AM Framework or simply the Framework) is the software entity that enables service availability by coordinating other software entities within a cluster.

The Availability Management Framework provides a view of one logical cluster that consists of a number of cluster nodes. These nodes host various resources in a distributed computing environment.

The Availability Management Framework provides a set of APIs to enable highly available applications. In addition to component registration and life cycle management, it includes functions for error reporting and health monitoring. The Availability Management Framework also assigns active or standby workloads to the components of an application as a function of component state and system configuration. The Availability Management Framework configuration allows prioritization of resources and provides for a variety of redundancy models. The Availability Management Framework also provides APIs for components to track the assignment of work or so-called component service instances among the set of components protecting the same component service instance.

3 System Description and System Model

This chapter presents the system description and the system model used by the SA Forum Application Interface Specification (AIS) of the Availability Management Framework (AMF).

An application that is managed by the Availability Management Framework to provide high levels of service availability must be structured into logical entities according to the model expected by the Framework. Furthermore, it must implement the state models and callback interfaces that allow the Framework to drive workload management, availability, and state management.

The following list shows the subjects treated in this chapter and the sections where they are described:

- Logical entities managed by the Availability Management Framework (Section 3.1)
- States and state models applicable to the relevant logical entities (Section 3.2 and Section 3.4)
- Fail-over and switch-over of service instances (Section 3.3)
- Component capability model (Section 3.5)
- Redundancy models supported by the Availability Management Framework (Section 3.6)
- Interactions between the component capability model and the redundancy models (Section 3.7)
- Dependencies among different entities (Section 3.8)
- Approaches for integrating legacy software and hardware entities in the framework (Section 3.9)
- Component monitoring (Section 3.10)
- Error detection, recovery, repair, and escalation policy (Section 3.11)

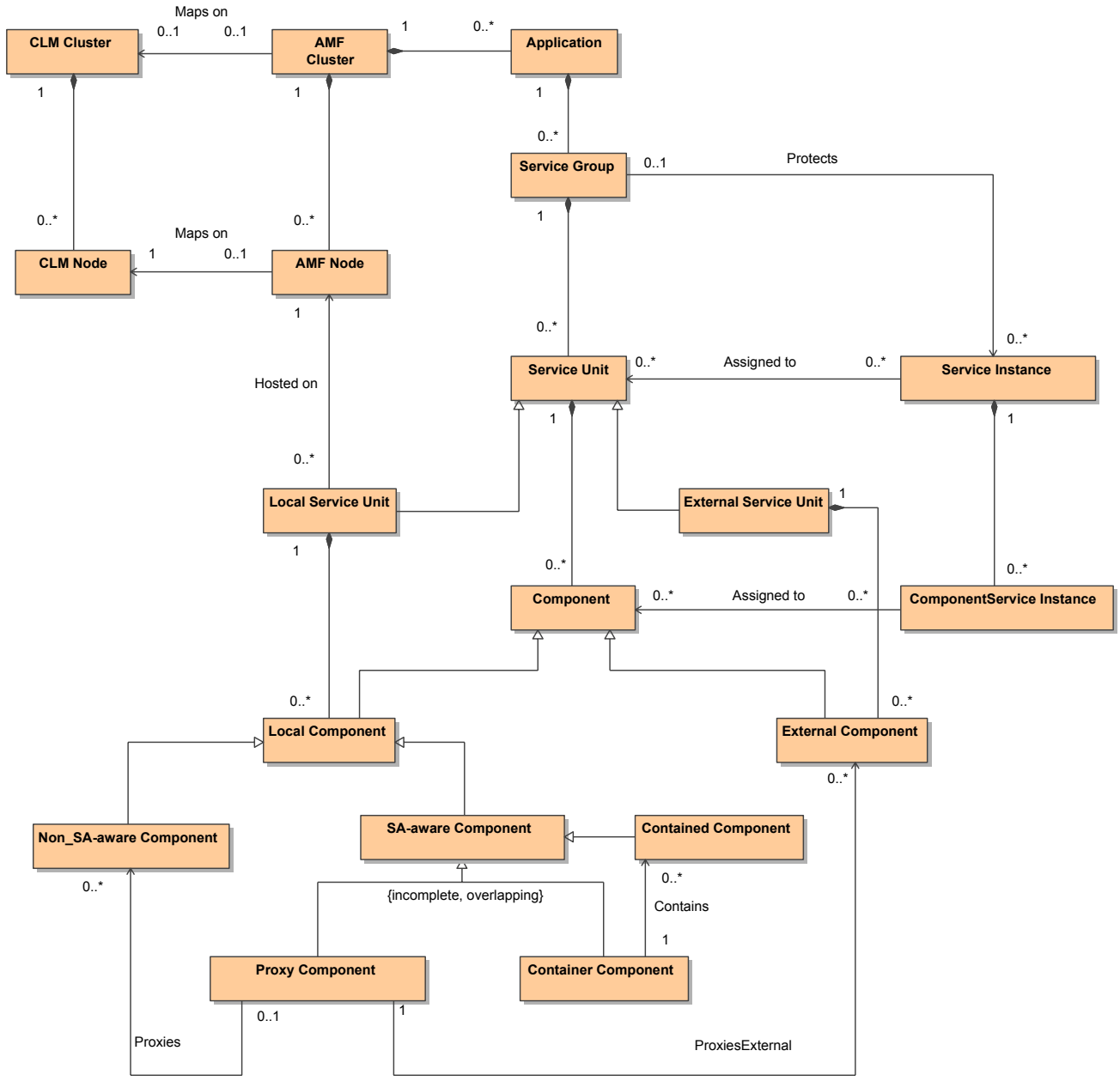
Note: The description of the Availability Management Framework configuration provides the pertinent attribute names, the names of object classes containing these attributes, and the sections containing the respective UML diagrams. Additional details on type, multiplicity, and values of these attributes are given in Chapter 8 and [7].
It is recommended to first read the entire Chapter 3 to fully understand the Availability Management Framework configuration described in these references.
Most entities of the Availability Management Specification have types, which are used to facilitate the configuration and for software management pur-

poses. These types are shortly described in a subsection of the sections describing the corresponding entities. Additional details are provided in Chapter 8 and [7].

3.1 Logical Entities

The Availability Management Framework uses an abstract system model to represent the resources under its control. This abstract model consists of various **logical entities** that are depicted in the UML diagram shown in FIGURE 1.

FIGURE 1 Availability Management Framework Logical Entities and Their Relations



Aggregation relationships in FIGURE 1 have multiplicities of '0..*' to account for situations in which the cluster configuration is being modified and some aggregations are temporarily empty.

Each logical entity of the system model is identified by a unique name.

All logical entities, their attributes, relationships, and mapping to the resources they represent are typically preconfigured and stored in a configuration repository. Dynamic modification of the system model is not precluded. The modeling and organization of this configuration information is described in Chapter 8. The access and modification of this configuration repository is provided by the Object Management interface of the IMM Service ([6]). It is assumed that the Availability Management Framework obtains the cluster configuration from the configuration repository and is notified of any changes.

The Availability Management Framework provides no API function to notify components about changes in the configuration repository.

3.1.1 Cluster and Nodes

Note: In the remainder of this document, the terms “CLM node”, “CLM cluster”, “AMF node”, and “AMF cluster” will be used as synonyms to Cluster Membership node and cluster and Availability Management Framework node and cluster, respectively.

3.1.1.1 AMF Nodes

The **AMF node** is a logical entity that represents a complete inventory of all Availability Management Framework entities on a **CLM node** (which is defined in [4]).

As shown in the UML diagram in Section 8.4, a CLM node can host at most one AMF node, and a PLM Execution Environment (abbreviated as EE, see the Platform Management Service specification [5]) can host at most one CLM node. These one-to-one relationships between an AMF node, its CLM node, and the CLM node’s PLM execution environment are administratively configured.

The configuration of an AMF node is valid even if

(a) no CLM node is mapped to the AMF node, or

(b) a CLM node is mapped to the AMF node, but the mapped CLM node is not in the cluster membership.

However, in both cases (a) and (b), the AMF node cannot be used to provide service, and none of the Availability Management Framework logical entities configured to be hosted by the AMF node can be instantiated.

An AMF node is also a logical entity whose various states are managed by the Availability Management Framework. Availability Management Framework administrative operations are defined for such nodes.

For a complete list of the attributes that are configured for an AMF node, refer to the description of the `SaAmfNode` UML class in Section 8.7 on page 344.

As there is a one-to-one association between an AMF node, its CLM node, and the PLM execution environment that hosts the CLM node, some notations have been adopted to make this specification more readable:

- Throughout the specification, when the word "node" is used without an explicit qualification, it means "AMF node".
- If "node" is used in the context of "joining the cluster", and "leaving the cluster", it actually means "the hosting CLM node". For example, the sentence fragment "when a node joins the cluster" should be interpreted as "when the CLM node hosting the AMF node joins the cluster".
- The reference to "the PLM execution environment that hosts an AMF node" actually means "the PLM execution environment that hosts the CLM node hosting the AMF node.
- The term "node reboot" should be interpreted as "a restart administrative operation on the PLM execution environment that hosts the CLM node hosting the AMF node".

3.1.1.2 AMF Cluster

The complete set of AMF nodes in the Availability Management Framework configuration defines the **AMF cluster**. For the relationship between the entities AMF cluster and CLM cluster, refer to the UML diagram in Section 8.4. Note that though the AMF cluster and the **CLM cluster** (defined in [4]) have a close relationship, they are not the same:

- It is possible that some AMF nodes may be mapped to some CLM nodes by configuration (see the `saAmfNodeClmNode` attribute of the `SaAmfNode` object class, shown in Section 8.7), whereas other AMF nodes are not mapped to configured CLM nodes, and thus do not provide service.
- During the life-span of the cluster, modifications may be made to the mapping of the AMF node to the CLM node.

- There may be nodes in the CLM cluster that are not meant to run software controlled by the Availability Management Framework. Thus, in a fully configured system, the CLM cluster may contain more nodes than the AMF cluster; in this case, the AMF cluster will be a proper subset of the CLM cluster.

The administrator is responsible for specifying the configuration for mapping AMF nodes to CLM nodes.

The AMF cluster is one of the entities that are under the Availability Management Framework's control, and its administrative state is managed by the Availability Management Framework (see Section 3.2.8). The Availability Management Framework defines certain administrative operations for the AMF cluster.

The Availability Management Framework knows the association of its nodes to the hosting PLM execution environments and shall use this association to initiate operations such as restarting the hosting PLM execution environment during recovery operations.

If a CLM node hosting an AMF node leaves the cluster membership, the CLM node is cleaned by the Availability Management Framework in the sense that no process belonging to AMF logical entities is left over on that node (see also Section 7.2.1 and Appendix D). Only persistent Availability Management Framework information will be available again when the node rejoins the cluster membership.

The Availability Management Framework can force a node to reboot while engaging certain recovery and repair mechanisms. During the reboot, the node leaves the cluster membership and rejoins it after successful initialization.

In contrast, the restart of an AMF node (see also Section 9.4.7) will only stop and start entities under the Availability Management Framework's control, without any impact on the cluster membership. The restart of the AMF cluster (see also Section 9.4.7) will restart all AMF nodes and will not affect the cluster membership. On the other hand, a **cluster reset** (see Section 7.4.7) restarts all PLM execution environments associated with all AMF nodes of the cluster, whereby the corresponding PLM execution environments are first terminated before any AMF node is instantiated again.

In the remainder of this specification, **cluster start** or **startup** is synonymous to the start of Availability Management Framework. The cluster start creates and instantiates the Availability Management Framework logical entities based on the Availability Management Framework configuration.

Applications to be made highly available are supposed to be configured in the Availability Management Framework configuration. Each application is configured to be hosted in one or more AMF nodes within the AMF cluster.

As there is a one-to-one association between an AMF cluster and its CLM cluster, some notations have been adopted to make the specification more readable:

- Throughout the specification, when the word "cluster" is used without an explicit qualification, it means "AMF cluster".
- If "cluster" is used in the context of "joining the cluster", and "leaving the cluster", it actually means "the associated CLM cluster". For example, the sentence fragment "when a node joins the cluster" should be interpreted as "when the CLM node associated with the AMF node joins the CLM cluster".

3.1.2 Components

A **component** is the logical entity that represents a set of resources to the Availability Management Framework. The resources represented by the component encapsulate specific application functionality. This set of resources can include hardware resources, software resources, or a combination of the two.

A component is the smallest logical entity on which the Availability Management Framework performs error detection and isolation, recovery, and repair. When deciding what is to be included in a component, the following two rules should be taken into account:

- The scope of a component must be small enough, so that a failure of the component has as little impact as possible on the services provided by the cluster.
- The component should include all functions that cannot be clearly separated for error containment or isolation purposes.

The Availability Management Framework associates the following states to a component: presence, operational, readiness, and HA. For more information on component states, refer to Section 3.2.2.

Resources that are contained—from a fault containment perspective—within the PLM execution environment hosting an AMF node are called **local resources**. This means that if the PLM execution environment fails, all of its local resources become inoperable. Local resources can be either software abstractions implemented by programs running within the PLM execution environment, or hardware equipment exclusively associated with the PLM execution environment (such as I/O devices).

All other resources are called **external resources**. For example, an intelligent I/O board in a blade chassis that is not dependent on a particular PLM execution environment to operate can be modeled as an external resource. A resource that is contained within a PLM execution environment hosting no AMF node is also modeled as an external resource, regardless of whether the execution environment hosts a CLM node.

The Availability Management Framework was primarily designed to manage local resources but it can also manage external resources. Unlike the case of local resources, the Availability Management Framework has little direct control over external resources. This difference justifies the distinction between two broad categories of components:

- **Local component**: a local component represents a subset of the local resources contained within the PLM execution environment hosting the AMF node.
- **External component**: an external component represents a set of external resources.

Section 3.1.2.1 up to Section 3.1.2.5 describe how the Availability Management Framework manages local and external components. The information provided includes:

- the notion of **component category** to distinguish components with different properties and different behavior. Two main categories of components are defined: Service Availability (SA)-aware (see Section 3.1.2.1) and non-SA-aware components (see Section 3.1.2.2);
- the notion of container and contained SA-aware components (see Section 3.1.2.1.1);
- the concepts of SA-aware proxy and non-SA-aware proxied components (see Section 3.1.2.3);
- the description of the component life cycle (see Section 3.1.2.4);
- the notion of component type (see Section 3.1.2.5).

3.1.2.1 SA-Aware Components

High levels of service availability can only be attained if errors are detected and isolated, a recovery is performed, and failed entities repaired efficiently. Faster error recovery is possible if components have been chosen or are written such that they can register and interact with the Availability Management Framework to implement specific workload assignments and recovery policies. Such components must be so designed that the Availability Management Framework can dynamically assign them workloads and choose the role in which the component will operate for each specific workload.

Only local components that are under the direct control of the Availability Management Framework can have such a high level of integration with this framework. Such components are termed **SA-aware** components.

Each SA-aware component includes at least one process that is linked to the Availability Management Framework library. One of these processes registers the component with the Availability Management Framework by invoking the `saAmfComponentRegister()` API function. This process, called the registered process for the component (for its definition, see Section 7.1.1) provides to the Availability Management Framework references to the availability control functions it implements. These control functions are implemented as callbacks.

Throughout the life of the component, the Availability Management Framework uses these control functions to direct the component execution by, for example:

- assigning workloads to the component,
- removing workloads from the component,
- and assigning the HA state to the component for each workload.

The registered process for a component executes the availability management requests it receives from these control functions and conveys such requests to other processes and to the hardware equipment of the local component, where necessary.

The registered process for a component may also provide the Availability Management Framework with feedback on its readiness to take an assignment for a particular workload.

Most control functions of the component can only be provided by the registered process; however, some control functions, such as healthcheck control functions, can be provided by any process of the component. The description of each API function presented in Chapter 7 explicitly mentions when the function is restricted to a registered

process for a component. Additionally, Appendix B contains a table showing which API or callback is restricted to registered processes for a component.

An SA-aware component has the following properties:

- its life cycle is directly controlled by the Availability Management Framework;
- each of its processes must exclusively belong to the component.

Note that container and contained components (which are discussed in Section 3.1.2.1.1) do not share all of the preceding properties.

When the context does not make the distinction apparent, the term **regular** SA-aware component is used to refer to an SA-aware component that is not contained in another component and does not implement any management function for assisting the Availability Management Framework in handling other components (such as proxy, container).

Legacy software, which runs on a node, and which was not initially designed as an SA-aware component can be converted to be SA-aware by adding a new process. This process acts as the registered process for the component, receives all management requests from the Availability Management Framework and converts them into specific actions on the legacy software using existing administration interfaces specific to the legacy software.

3.1.2.1.1 Container and Contained Components

This section describes the particular properties of container and contained components. As other features of container and contained components are described in other sections of this and other chapters, Chapter 6 summarizes the corresponding information and also provides additional information.

Purpose

The concept of **container** and **contained** components allows the Availability Management Framework to integrate components that are not executed directly by the operating system, but rather in a controlled environment running on top of the operating system. Widespread environments are runtime environments, virtual machines, or component frameworks.

Properties of Container Components

- The main task of the container component is to cooperate with the Availability Management Framework to handle the life cycle of contained components. The following definitions are used in this explanation and throughout this document:
  - ⇒ The container component that handles the life cycle of a contained component in cooperation with the Availability Management Framework is termed the **associated container component** to the contained component.
  - ⇒ Conversely, a contained component whose life cycle is handled by a container component in cooperation with the Availability Management Framework is termed an **associated contained component** to the container component.
  - ⇒ For ease of expression when referring to a contained component, the term **collocated contained component** is used to refer to a contained component that has the same associated container component.

  Which actions are performed by the associated container component and by the Availability Management Framework for handling the life cycle of a contained component is explained in detail in Section 6.2.
  The interactions between contained components and the associated container to implement the life cycle of the contained components are not defined by the Availability Management Framework specification.
- A single container component can be the associated container component of various contained components.
- A container component and all its associated contained components must reside on the same AMF node.
- The life cycle of a container component is directly controlled by the Availability Management Framework.
- The termination of a container component (for instance, in case of a failure of the component) implies the termination of all associated contained components (see also Section 6.3). In this sense, a container component “contains” the associated contained component.
- A process belonging to a container component can also belong to its associated contained components.

Properties of Contained Components

- The life cycle of contained components is handled by the Availability Management Framework in cooperation with the associated container component (see also Section 6.2).
- The termination of a contained component does not imply the termination of either the associated container component or of the collocated contained components (see also Section 6.3).
- A process belonging to a contained component belongs also to its associated container component and may also belong to some of its collocated contained components.

3.1.2.2 Non-SA-Aware Components

Components that do not register directly with the Availability Management Framework are called **non-SA-aware** components. However, such components may have processes linked with the Availability Management Framework Library.

Typically, non-SA-aware components are registered with the Availability Management Framework by dedicated SA-aware components that act as proxies between the Availability Management Framework and the non-SA-aware components. These dedicated SA-aware components are called **proxy components**. The components for which a proxy component mediates are called **proxied components**. Proxy and proxied components are explained in more detail in Section 3.1.2.3.

3.1.2.2.1 External Components

To keep maximum flexibility in the way external resources interact with nodes, which is often device-dependent or proprietary, the Availability Management Framework does not interact directly with external components and manages external components always as proxied components.

3.1.2.2.2 Non-Proxied, Non-SA-Aware Components

The Availability Management Framework supports both proxied and non-proxied, non-SA-aware local components. For non-proxied, non-SA-aware local components, the role of the Availability Management Framework is limited to the management of the component life cycle. The Availability Management Framework instantiates a non-proxied, non-SA-aware component when the component needs to provide a service and terminates this component when the component must stop providing the service. Processes of a local non-SA-Aware component must exclusively belong to that component.

3.1.2.2.3 Integration and Usage of Non-SA-Aware Local Components

Application developers are encouraged to design applications that will run on nodes as a set of SA-aware components registered directly with the Availability Management Framework; however, non-SA-aware local components may be used instead for the following reasons:

- Some system resources such as networking resources or storage resources are implemented by the operating environment, and their activation or deactivation is usually performed by running administrative command line interfaces. No actual process is needed to implement these resources, and requiring the implementation of a registering process for such resources adds unnecessary complexity.
- For components representing only local hardware resources, making these components SA-aware components with a registering process adds unnecessary complexity.
- Existing clustering products support looser execution models than the execution model of SA-aware components. For these products, the integration between the applications and the clustering middleware is minimal: the clustering middleware is only responsible for starting, stopping, and monitoring applications, but does not expose APIs for finer-grained control of the application in terms of workload and availability management.
  It is important to facilitate the migration of third party products from these existing clustering products to products providing the Availability Management Framework interfaces without requiring the transformation of these third party products into SA-aware components.
- Some complex applications such as databases or application servers already provide their own availability management for their various building blocks. When moving these applications under the Availability Management Framework's control, different functions can be modeled as separate components; however, some controlling entity within the application might still be interposed between the Availability Management Framework and the individual components. The concept of the proxy component can be used in this case as an interposition layer between the Availability Management Framework and all other components of the application.

3.1.2.3 Proxy and Proxied Components

The Availability Management Framework uses the availability control functions registered by a proxy component to control the proxy component and the proxied components for which the proxy component mediates.

The proxy component is an SA-aware component that is responsible for conveying requests made by the Availability Management Framework to its proxied components. A contained component must not be a proxy component.
The interactions between proxied components and their proxy component are private and not defined by this specification.

The Availability Management Framework determines the proxied components for which a proxy component is responsible when the proxy component registers with the framework, based on configuration and other factors like availability of components in the cluster. The Availability Management Framework conveys this decision to the proxy component by assigning it a workload in the form of a component service instance (for the definition of component service instance, see Section 3.1.3).

The proxy component registers proxied components with the Availability Management Framework; however, the proxied components are independent components as far as the Availability Management Framework is concerned. As such, if a proxy component fails, or an entity containing it is prevented by the administrator from providing service, another component (usually the component acting as standby to the failed proxy component) can register the proxied component again. This new proxy component assumes then the mediation for the failed component without affecting the service provided by the proxied component. If no proxy component is available to take over the mediation service for the proxied component, the Availability Management Framework loses control of the proxied component and becomes unaware of whether the proxied component is providing service.

As various other features of proxy and proxied components are described in various sections of this and other chapters, Chapter 5 summarizes the information on all these features and also provides additional information. However, for convenience of the reader, the following notes list some key features:

- A single proxy component can mediate between the Availability Management Framework and multiple proxied components.
- The redundancy model (for a discussion of this notion, refer to Section 3.6) of the proxy component can be different from that of its proxied components.

- The Availability Management Framework does not consider the failure of the proxied component to be the failure of the proxy component. Similarly, the failure of the proxy component does not indicate a failure of the proxied components (see Section 5.3).
- No process of a proxied component registers with the Availability Management Framework. A proxied component is registered by a process of the proxy component 'proxying' this proxied component.
- The process of the proxy component registered for a proxied component mediates all the interactions between the Availability Management Framework and the proxied component it is registered for.
- One process of a proxy component registers the proxy component itself. The same or another process of the proxy component may register a proxied component whenever the proxy component is 'proxying' this proxied component.

3.1.2.4 Component Life Cycle

The Availability Management Framework directly controls the life cycle of non-proxied, local components through a set of command line interfaces that must be provided by each component.

The Availability Management Framework indirectly controls the life cycle of proxied components through their proxies. However, command line interfaces may also be used by the Availability Management Framework to control some aspects of the life cycle of local proxied components.

For information about command line interfaces for the local component life cycle management, refer to Chapter 4.

The Availability Management Framework distinguishes between two categories of components in its life cycle management:

- **pre-instantiable components**: such components have the ability to stay idle when they get instantiated by the Availability Management Framework. They start to provide a particular service only when instructed to do so (directly or indirectly) by the Availability Management Framework. The Availability Management Framework can speed up recovery and repair actions by keeping a certain number of pre-instantiated components, which can then take over faster the work of failed components. All SA-aware components are pre-instantiable components.
- **non-pre-instantiable components**: such components provide service as soon as they are instantiated. Hence, the Availability Management Framework cannot instantiate them in advance as spare entities. All non-proxied, non-SA-aware components are non-pre-instantiable components.

The following table shows the various component categories and subcategories.

Table 3 Component Categories

Locality	HA Awareness	Proxy Property	Life Cycle Management
local	regular SA-aware	non-proxy	pre-instantiable
local	(SA-aware) proxy	proxy	pre-instantiable
local	(SA-aware) container	proxy or non-proxy	pre-instantiable
local	(SA-aware) contained	non-proxy	pre-instantiable
local	non-SA-aware	non-proxied	non-pre-instantiable
local	non-SA-aware	proxied	pre-instantiable or non-pre-instantiable
external	non-SA-aware	proxied	pre-instantiable or non-pre-instantiable

3.1.2.5 Component Type

The Availability Management Framework supports the notion of a **component type**. A component type represents a particular version of the software or hardware implementation which is used to construct components. All components of the same type share the attribute values defined in the component type configuration. Some of the attribute values may be overridden, and some of them may be extended in the component configuration.
Details on the configuration of a component type and of a component are provided in Section 8.13 on page 358 and in [7].

3.1.3 Component Service Instance

A **component service instance** (CSI) represents the workload that the Availability Management Framework can dynamically assign to a component. High availability (HA) states are assigned to a component on behalf of its component service instances. The Availability Management Framework chooses the HA state of a component for each particular component service instance, as described in Section 3.2.2.4. To help AMF to make this choice more comprehensive, some components may provide information on their readiness to assume a particular component service instance (see Section 3.2.2.5).

Each component service instance has a set of attributes (**name/value pairs**), which characterize the **workload** assigned to the component. Several attributes with the same name may appear in the set of attributes of a component service instance, thus providing support for multivalued attributes. These attributes are not used by the Availability Management Framework and are just passed to the components.

The Availability Management Framework supports the notion of proxy CSI. A **proxy CSI** represents the special workload of 'proxying' a proxied component.
A proxied component must be configured with the proxy CSI that provides 'proxying'. The Availability Management Framework configuration specifies to which proxy components this proxy CSI can be assigned.
Note that a proxy component can be configured to have multiple CSI assignments, one or more for handling proxied components and others for providing other services. In terms of functionality, there is no difference between a proxy CSI corresponding to the workload of 'proxying' proxied components and CSI assignments corresponding to the workload of other services.

The Availability Management Framework supports the notion of **container CSI**. A container CSI represents the special workload of managing the life cycle of contained components.
A contained component must be configured with the container CSI. The Availability Management Framework determines, based on its configuration, the container components to which this container CSI is assigned. Which of these container components will become the associated container is explained in detail in Section 6.2.

The container CSI can contain information to be passed by the associated container component to the corresponding contained component. How this information is passed is a private interface between container and contained components.

Note that a container component can be configured to have multiple CSI assignments, one or more for handling contained components, and others for providing other services. In terms of functionality and syntax, there is no difference between a container CSI used to determine the associated container component and other CSIs corresponding to the workload of other services.

3.1.3.1 Component Service Type

The Availability Management Framework supports the notion of **component service type**.
The component service type is the generalization of similar component service instances (that is, similar workloads) that are seen by the Availability Management Framework as equivalent and handled in the same manner. The configuration of a component indicates which component service types the component supports. These component service types must be chosen from the set of component service types supported by the component type to which the component belongs.
The component service type defines the list of the attribute names for all component service instances belonging to the type.
Details on the configuration of a component service type and of a component service instance are provided in Section 8.12 on page 356 and in [7].

3.1.4 Service Unit

A **service unit** (SU) is a logical entity that aggregates a set of components combining their individual functionalities to provide a higher level service. Aggregating components into a logical entity managed by the Availability Management Framework as a single unit provides system administrators with a simplified, coarser-grained view. Most administrative operations apply to service units as opposed to individual components.

A service unit can contain any number of components, but a particular component can be configured in only one service unit. The components that constitute a service unit can be developed in isolation, and a component developer might be unaware of which components constitute a service unit. The service units are defined at deployment time.

As a component is always enclosed in a service unit, from the Availability Management Framework's perspective, the service unit is the unit of redundancy in the sense that it is the smallest logical entity that can be instantiated in a redundant manner (that is, more than once).

The Availability Management Framework associates presence, administrative, operational, readiness, and HA states to service units (latter on behalf of service instances). Each of these states, with the exception of the administrative state, represents an aggregated view of the corresponding state of each component within the service unit. The rules applied to obtain these aggregated states are specific to each state and are described in Section 3.2.

Local components and external components cannot be mixed within a service unit. The Availability Management Framework distinguishes between **local service units** and **external service units**. Local service units can contain only local components (they are collocated on the same node). External service units can contain only external components. The external components represent resources that are external to the cluster.

A proxy component and its non-pre-instantiable proxied component can reside in the same or in different service units; however, a proxy component and its pre-instantiable proxied component must not reside in the same service unit in order to prevent cyclic dependencies during the instantiation of the service unit. If the proxy and proxied local components are hosted in different service units, these service units may reside on different nodes.

In a service unit, contained components must not be mixed with components of other categories. The rationale for this decision is explained in Section 6.1.5.

All contained components in a service unit must have the same associated container component, and this association is achieved by the usage of a single container CSI (see also Section 6.2).

A service unit that contains at least one pre-instantiable component is called a **pre-instantiable service unit**; otherwise, it is called a **non-pre-instantiable service unit**.

3.1.4.1 Service Unit Type

The Availability Management Framework supports the notion of a **service unit type**. The service unit type defines a list of component types and, for each component type, the number of components that a service unit of this type may accommodate. A service unit of a given type may only consist of components of the component types from that list, and the number of these components must be within the range specified for the component type. All service units of the same type share the attribute values defined in the service unit type configuration. Some of the attribute values may be overridden in the service unit configuration.
All service units of the same type can be assigned service instances derived from the same set of service types.
Details on the configuration of a service unit type and of a service unit are provided in Section 8.10 on page 350 and in [7].

3.1.5 Service Instances

In the same way as components are aggregated into service units, the Availability Management Framework supports the aggregation of component service instances into a logical entity called a **service instance** (SI). A service instance aggregates all component service instances to be assigned to the individual components of the service unit in order for the service unit to provide a particular service.

A service instance can contain multiple component service instances, but a particular component service instance can be configured in only one service instance.

A service instance represents a single workload assigned to the entire service unit.

When a service unit is available to provide service (in-service readiness state, see Section 3.2.1.4), the Availability Management Framework can assign HA states to the service unit for one or more service instances. When a service unit becomes unavailable to provide service (out-of-service readiness state), the Availability Management Framework removes all service instances from the service unit. A service unit might be available to provide service but not have any assigned service instance.

The Availability Management Framework assigns a service instance to a service unit programmatically by assigning each individual component service instance of the service instance to a specific component within the service unit.

The assignment of the component service instances of a service instance to the components of a service unit takes into account the type of component service instance supported by each component. A component service instance can be assigned to a given component only if the component configuration indicates that the component supports this particular type of component service instance, the component configuration permits the assignment of at least one more component service instance of this type, and the component is ready to assume (it did not indicate otherwise) the workload associated with the component service instance. When a service instance contains several component service instances of the same type, this specification does not dictate how, within the service unit, the Availability Management Framework assigns them to the components that support this particular type. This choice is implementation-defined.

The number of component service instances aggregated into a service instance may differ from the number of components aggregated into the service unit to which the service instance is assigned. Some components may be left without any component service instance assignment whereas other components may have several component service instances assigned to them.

3.1.5.1 Service Type

The Availability Management Framework supports the notion of a **service type**.
The service type defines a list of component service types of which a service instance may be composed. The service type also defines for each component service type the number of component service instances that a service instance of the given type may aggregate. All service instances of the same type share the attribute values defined in the service type configuration.
Details on the configuration of a service type and of a service instance are provided in Section 8.11 on page 353 and in [7].

3.1.6 Service Groups

To ensure service availability in case of component failures, the Availability Management Framework manages redundant service units.
A **service group** (SG) is a logical entity that groups one or more service units in order to provide service availability for a particular set of service instances. Any service unit of the service group must be able to take an assignment for any service instance of this set. Furthermore, to participate in a service group, all components in the service unit must support the capabilities required for the **redundancy model** defined for the service group.
The redundancy model defines how the service units in the service group are used to provide service availability. For details about service group redundancy models, refer to Section 3.6.

Note: For readability purposes, and if the context permits, this document uses expressions like “the components of a service group“ to mean “the components of service units participating in the service group”.

3.1.6.1 Service Group Type

The Availability Management Framework supports the notion of a **service group type**.
The service group type is a generalization of similar service groups that follow the same redundancy model, provide similar availability, and are composed of units of the same service unit types. All service unit types defined in the service group type must be capable of supporting a common set of service types. All service groups of the same type share the attribute values defined in the service group type configuration. Some of the attribute values may be overridden in the service group configuration.
Details on the configuration of a service group type and of a service group are provided in Section 8.9 on page 348 and in [7].

3.1.7 Application

An **application** is a logical entity that contains one or more service groups. An application combines the individual functionalities of the constituent service groups to provide a higher level service.

This aggregation provides the Availability Management Framework with a further scope for fault isolation and fault recovery.

From a software administration point of view, this grouping into application reflects the set of service units and their components, which are delivered as a consistent set of software packages, which results in tighter dependency with respect to their upgrade.

An application can contain any number of service groups, but a given service group can be configured in only one application.

Dependencies amongst service instances (described in Section 3.8.1 on page 185) are more common amongst service instances belonging to the application than amongst service instances of different applications.

3.1.7.1 Application Type

The Availability Management Framework supports the notion of an **application type**. An application type defines a list of service group types, which implies that an application of the given type must be composed of service groups of types from that list. All applications of the same type share the attribute values defined in the application type configuration.

Details on the configuration of an application type and of an application are provided in Section 8.8 on page 346 and in [7].

3.1.8 Protection Groups

A **protection group** for a specific component service instance is the group of components to which the component service instance has been assigned. The name of a protection group is the name of the component service instance that it protects.

A protection group is a dynamic entity, which changes when component service instances are assigned to components or removed from components.

3.1.9 Mapping of Service Units to Nodes

Service groups and service units have an optional **node group configuration attribute**. A node group just contains a list of nodes.

Service units have an optional configuration attribute (`saAmfSUHostNodeOrNodeGroup` in the `SaAmfSU` object class, shown in Section 8.10), which can either represent a node or a node group. The service unit can only be instantiated on the node (if a node is specified) or on one of the nodes of the node group (if a node group is configured).

The Availability Management Framework maps each service unit onto a node at the time the service unit is introduced to the cluster (that is, at cluster startup or when the service unit is added to the configuration), and this mapping persists until the service unit is removed from the configuration, or the cluster is restarted. In other words, the node group does not provide an additional level of protection against node failures.

When the Availability Management Framework decides to instantiate a local service unit in accordance with the pertinent redundancy model, it performs the following checks:

1. If a node is configured for the service unit, the service unit will be instantiated on this node.
2. If instead a node group is configured for the service unit, the Availability Management Framework selects a node from the node group using an implementation-specific policy to instantiate the service unit on this node.
3. If no node or node group is configured for the service unit, the Availability Management Framework checks whether a node group is configured for the service group (`saAmfSGSuHostNodeGroup` attribute in the `SaAmfSG` object class, shown in Section 8.9).
4. If a node group is configured for the service group, the Availability Management Framework selects a node from the node group using an implementation-specific policy to instantiate the service unit on this node.
5. If no node group is configured for the service group, the Availability Management Framework selects any node using an implementation-specific policy to instantiate the service unit on it.

If node groups are configured for both the service units of a service group and the service group, the nodes contained in the node group for the service unit can only be a subset of the nodes contained in the node group for the service group. If a node is configured for a service unit, it must be a member of the node group for the service group, if configured.

It is an error to define the `saAmfSUHostNodeOrNodeGroup` attribute for an external service unit. It is also an error to define the `saAmfSGSuHostNodeGroup` attribute if a service group contains only external service units.

Section 6.1.5 provides additional rules on the configuration of node and node groups for service units containing contained components and for the service groups containing these service units, so that this configuration aligns with the configuration of nodes and node groups for service units containing container components and for the service groups containing these service units.

3.1.10 Service Unit Instantiation

When the Availability Management Framework instantiates a pre-instantiable service unit, it:

- runs the `INSTANTIATE` command (see Section 4.6) for SA-aware components (excluding contained components),
- invokes the `saAmfContainedComponentInstantiateCallback()` callback (see Section 7.10.4) of the associated container component for each contained component of the service unit,
- invokes the `saAmfProxiedComponentInstantiateCallback()` callback (see Section 7.10.2) of the proxies of all pre-instantiable proxied components of the service unit,
- and performs no action for non-pre-instantiable components. Such components are instantiated during the assignment of service instances to the service unit (see Section 3.2.2.4 on page 77).

When the Availability Management Framework instantiates a non-pre-instantiable service unit, it:

- invokes the `saAmfCSISetCallback()` callback (see Section 7.9.2) of the proxies of all proxied components of the service unit and
- runs the `INSTANTIATE` command (see Section 4.6) for all non-proxied components.

Note that this processing creates an implicit inter-service unit dependency, as the Availability Management Framework needs to instantiate the service units containing proxy components (and sometimes even assign them an active HA state for a service instance) before the instantiation of service units containing proxied components can be successfully completed.

3.1.11 Illustration of Logical Entities

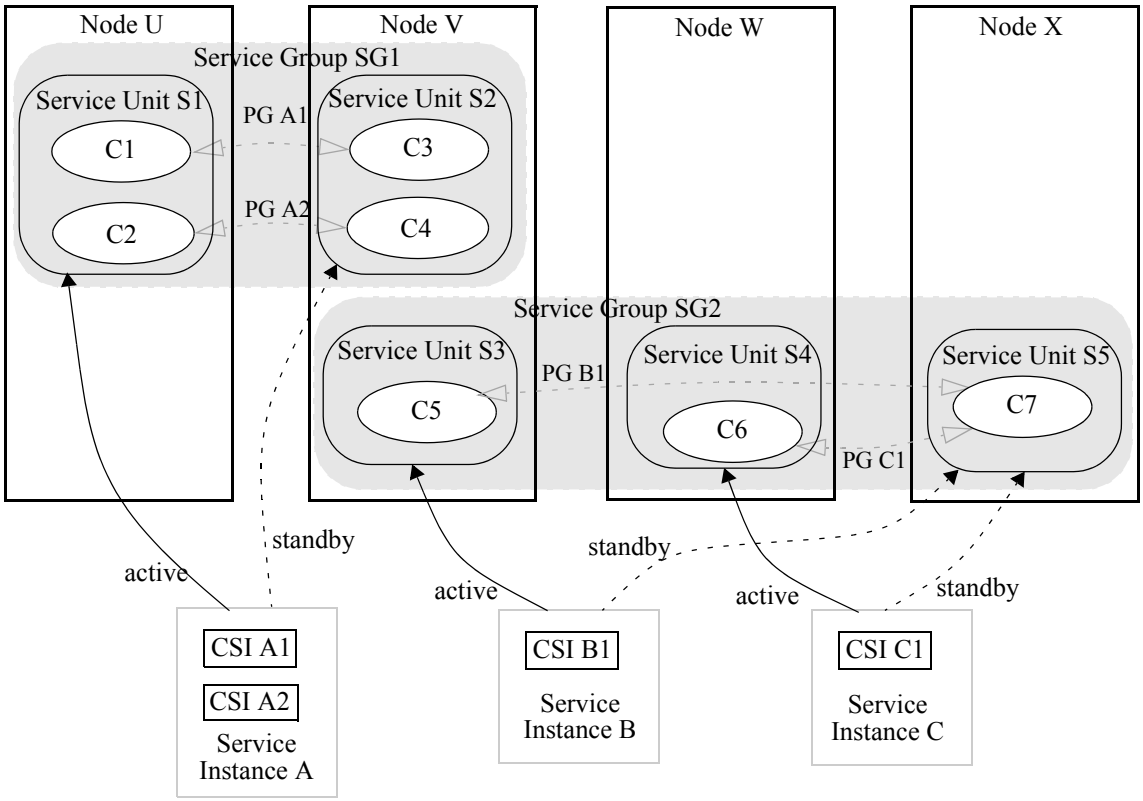
The example in FIGURE 2 shows two service groups, SG1 and SG2. SG1 supports a single service instance (A) and SG2 supports two service instances (B and C).

On behalf of service instance A, service unit S1 is assigned the active HA state and service unit S2 the standby HA state.

Each of the service units S1 and S2 contains two components. The component service instance A1 is assigned to the components C1 and C3, and the component service instance A2 is assigned to the components C2 and C4. Two protection groups A1 and A2 are created, with protection group A1 containing components C1 and C3 and protection group A2 containing components C2 and C4. Note that the name of the protection group is the same as the name of the component service instance. Thus, protection group A1 contains the components that support component service instance A1.

On behalf of service instance B, service unit S3 is assigned the active HA state and service unit S5 the standby HA state. Similarly, on behalf of service instance C, service unit S4 is assigned the active HA state and service unit S5 the standby HA state. Each of these service units contains a single component (C5, C6, C7). Thus, while components C5 and C6 are assigned the active HA state for only single component service instances (B1 and C1, respectively), component C7 is assigned the standby HA state for two component service instances (B1 and C1). Two protection groups (B1 and C1) are created, with protection group B1 containing components C5 and C7 and protection group C1 containing components C6 and C7.

FIGURE 2 Elements of the System Model



3.2 State Models

The following sections describe the different states associated with service units, components, service instances, component service instances, service groups, applications, and nodes. The Availability Management Framework API provides state management only for components and component service instances by using a subset of the states described in the following subsections. The other states included in the state model are relevant for System Management as well as for a clear definition and extension of this specification.

3.2.1 Service Unit States

In some cases when describing the properties and states of service units, references are made to properties and states of a node or cluster containing it. For readability reasons, it is not always mentioned that these references, obviously, only apply to local service units and are to be ignored for external service units.

3.2.1.1 Presence State

The **presence state** is supported at the service unit and component levels and reflects the component life cycle. It takes one of the following values:

- uninstantiated
- instantiating
- instantiated
- terminating
- restarting
- instantiation-failed
- termination-failed

First, the presence state of a non-pre-instantiable service unit is considered:

Note that the presence state of a service unit is described in this section in terms of the presence state of its constituent components, which is explained in detail in Section 3.2.2.1.

When all components are uninstantiated, the service unit is **uninstantiated**. When the first component moves to instantiating, the service unit also becomes **instantiating**.

When all pre-instantiable components of a service unit enter the instantiated state, the service unit becomes **instantiated**.
A non-pre-instantiable service unit is instantiated if it has successfully been assigned the active HA state on behalf of a service instance (see Section 3.2.1.5). Note that a non-pre-instantiable service unit may be assigned one and only one service instance.
If, after all possible retries, a component cannot be instantiated, the presence state of the component is set to instantiation-failed, and the presence state of the service unit is also set to **instantiation-failed**. If some components are already instantiated when the service unit enters the instantiation-failed state, the Availability Management Framework terminates them. These components will enter either the uninstantiated state if they are successfully terminated or the termination-failed state if the Availability Management Framework was unable to terminate them correctly (refer also to Section 4.7 and Section 4.8).

When the first component of an already instantiated service unit becomes terminating, the service unit becomes **terminating**. If the Availability Management Framework fails to terminate a component, the presence state of the component is set to termination-failed and the presence state of the service unit is also set to **termination-failed**.

When all components enter the restarting state, the service unit become **restarting**. However, if only some components are restarting, the service unit is still instantiated.

The management of the presence state of a pre-instantiable service unit is very similar to what was previously described for a non-pre-instantiable service unit, except that a pre-instantiable service unit becomes instantiated or terminating based only on the presence state of its pre-instantiable components; when all pre-instantiable components within a pre-instantiable service unit are instantiated, the service unit becomes instantiated. If any errors occur when instantiating any of the constituent components of the service unit, the presence state of the service unit becomes instantiation-failed. Similarly, if errors occur when terminating any of the constituent components of the service unit, its presence state becomes termination-failed.

3.2.1.2 Administrative State

The administrative state of a service unit is an extension of the administrative state proposed by the ITU X.731 state management model ([9]). The administrative state of a service unit can be set by the system administrator.
The administrative state of a service unit as well as the administrative states of the service group (see Section 3.2.5), the node (see Section 3.2.6.1), the application containing it (see Section 3.2.7), and the cluster (see Section 3.2.8) enable the Availability Management Framework to determine whether the service unit is administratively allowed to provide service.

Valid values for the administrative state of a service unit are:

- **unlocked**: the service unit has not been directly prohibited from taking service instance assignments by the administrator.
- **locked**: the administrator has prevented the service unit from taking service instance assignments.
- **locked-instantiation**: the administrator has prevented the service unit from being instantiated by the Availability Management Framework; the service unit is then not instantiable.
- **shutting-down**: the administrator has prevented the service unit from taking new service instance assignments and requested that existing service instance assignments be gracefully removed. When all service instances assigned to the service unit have finally been removed, its administrative state becomes locked.

The administrative state of a service unit is one of the states that determine the readiness state (see Section 3.2.1.4) of that service unit.

The administrative state of a service unit is persistent even when all nodes within the cluster are rebooted.

The administrative state of a service unit is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state has an impact on component service instance assignments.

3.2.1.3 Operational State

The **operational state** of the service unit reflects its error status. Valid values for the operational state of a service unit are:

- **enabled**: the operational state of a service unit transitions from disabled to enabled when a successful repair action has been performed on the service unit (see Section 3.11.1.4).

- **disabled**: the operational state of a service unit transitions to disabled if a component of the service unit has transitioned to the disabled state and the Availability Management Framework has taken a recovery action at the level of the entire service unit.

It is the Availability Management Framework that determines the value for the operational state of a service unit.

A service unit is enabled when the node containing this service unit joins the cluster for the first time. It is set to disabled when a fail-over recovery is executed within its scope, or if its presence state is set to instantiation-failed or termination-failed. After a successful repair, it is set again to enabled by the entity performing the repair (Availability Management Framework or other entity). An administrative operation is provided to clear the disabled state of a service unit, so that an entity other than the Availability Management Framework can perform the repair and declare the service unit repaired. When a restart recovery is executed in the scope of a service unit, the restart is considered as an instantaneous, combined recovery and repair action; therefore, the operational state of the service unit remains enabled in such cases. The operational state of the service unit is also re-evaluated whenever the operational state of one of its components transitions from disabled to enabled as a result of clearing an error condition (see Section 7.12.2).

3.2.1.4 Readiness State

The operational, administrative, and presence states of a service unit, the operational state of its containing node, and the administrative states of its containing node, service group, application, and the cluster are combined into another state, called the **readiness state** of a service unit. This state indicates if a service unit is eligible to take service instance assignments from an administrative and health status viewpoint. This state is used by Availability Management Framework to decide whether a service unit is eligible to receive service instance assignments.

The readiness state of a service unit is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state has an impact on component service instance assignments.

Valid values for the readiness state of a service unit are:

⇒ **out-of-service**

The readiness state of a **non-pre-instantiable service unit** is out-of-service if one or more of the following conditions are met:

- its operational state or the operational state of its containing node is disabled;
- its administrative state or the administrative state of its containing service group, AMF node, application, or the cluster is either locked or locked-instantiation;
- the CLM node to which the containing AMF node is mapped is not a member.

The readiness state of a **pre-instantiable service unit** is out-of-service if

- any of the preceding conditions that cause a non-pre-instantiable service unit to become out-of-service is true,
- or its presence state is neither instantiated nor restarting,
- or the service unit contains contained components, and their configured container CSI is not assigned active or quiescing to any container component on the node that contains the service unit.

When the readiness state of a service unit is out-of-service, no new service instance can be assigned to it. If service instances are already assigned to the service unit at the time when the service unit enters the out-of-service state, they are transferred to other service units (if possible) and removed.
Note that in some cases, pre-instantiable service units may be instantiated while they are out-of-service. However, non-pre-instantiable service units are terminated when they transition to the out-of-service readiness state.

⇒ **in-service**

The readiness state of a **non-pre-instantiable service unit** is in-service if all of the following conditions are met:

- its operational state and the operational state of its containing node is enabled;
- its administrative state and the administrative states of its containing service group, AMF node, application, and the cluster are unlocked;
- the CLM node to which the containing AMF node is mapped is a member node.

The readiness state of a **pre-instantiable service unit** is in-service if

- all of the preceding conditions that cause a non-pre-instantiable service unit to become in-service are true,
- and its presence state is either instantiated or restarting,
- and the configured container CSI of all contained components of the service unit is assigned active to at least one container component on the node that contains the service unit.

When a service unit is in the in-service readiness state, it is eligible for service instance assignments; however, it is possible that it has not yet been assigned any service instance.

⇒ **stopping**

The readiness state of a service unit is stopping if all of the following conditions are met:

- its operational state and the operational state of its containing node is enabled,
- none of the administrative states of itself, the containing service group, AMF node, application, CLM node, or the cluster is locked or locked-instantiation,
- at least one of the administrative states of itself, the containing service group, AMF node, application, CLM node, or the cluster is shutting-down, or the container component which is handling the life cycle of contained components of the service unit has the quiescing HA state for the container CSI of the contained components, and
- the CLM node to which the containing AMF node is mapped is a member node.

When a service unit is in the stopping state, no service instance can be assigned to it, but already assigned service instances are not removed until the service unit's components indicate to do so.

Table 4 shows how a pre-instantiable service unit's readiness state is derived from the operational state, the presence state, and the administrative states of itself, and the administrative states of its enclosing AMF node, service group, application, and AMF cluster. The same table applies to non-pre-instantiable service units by ignoring the “Service Unit’s Presence State column and assuming that the containing CLM node is in the cluster membership in the first two rows and regardless of whether the CLM node is or not in the cluster membership for the third row.

Table 4 Service Unit's Readiness State

Cluster's Administrative State	Application's Administrative State	Service Unit's Administrative State	Service Group's Administrative State	AMF Node's Administrative State	AMF Node's Operational State	Service Unit's Operational State	Service Unit's Presence State	Service Unit's Readiness State
unlocked	unlocked	unlocked	unlocked	unlocked	enabled	enabled	instantiated or restarting.	in-service
One or more columns contain the shutting-down state, and none is locked or locked-instantiation.					enabled	enabled	instantiated, instantiating, terminating, or restarting	stopping
All other combinations of locked/locked-instantiation/unlocked/shutting-down, enabled/disabled and any presence state.								out-of-service

3.2.1.5 HA State of a Service Unit for a Service Instance

When a service instance is assigned to a service unit, the Availability Management Framework assigns an **HA state** to the service unit for that service instance. The HA state takes one of the following values:

- **active**: the service unit is currently responsible for providing the service characterized by this service instance.
- **standby**: the service unit acts as a standby for the service characterized by this service instance.
- **quiescing**: the service unit that had previously an active HA state for this service instance is in the process of quiescing its activity related to this service instance. In accordance with the semantics of the shutdown administrative operations, the quiescing is performed by rejecting new users of the service characterized by this service instance while still providing the service to existing users until they all terminate using it. When no user is left for that service, the components of the service unit indicate that fact to the Availability Management Framework, which transitions the HA state to quiesced.
  The quiescing HA state is assigned as a consequence of a shutdown administrative operation.
- **quiesced**: the service unit that had previously an active or quiescing HA state for this service instance has now quiesced its activity related to this service instance, and the Availability Management Framework can safely assign the active HA state for this service instance to another service unit.

The quiesced state is assigned in the context of switch-over situations (for a description of switch-over, refer to Section 3.3).

Note: At any point of time, a service unit may have multiple service instance assignments.

The service units do not have an HA state of their own. They are assigned HA states on behalf of service instances.

Note: In the remainder of the document, the usage of the terminology “active or standby service units”, without mentioning for which service instance or service instances the service unit has been assigned a particular HA state, will be deemed legal when the context makes it obvious. This terminology is mostly applicable in scenarios in which all service instances assigned to a particular service unit share the same HA state and the service unit is incapable of sustaining a mix of HA states for the assigned service instances.

For simplicity of expression, the term **active assignment of/for a service instance** (or simply **active assignment** if the context makes it clear which service instance is meant) is used to mean the assignment of the active HA state to a service unit for this service instance. Similar terms are also used for the other HA states, such as **standby assignment**.

Taking into consideration the configuration of each service group (list of service instances, list of service units, redundancy model attributes, and so on) and the current value of the administrative and operational states of their service units and service instances, the Availability Management Framework dynamically assigns the HA state to the service units for the various service instances. Section 3.6 describes how these assignments are performed for the various redundancy models.

Though some aspects differ from one redundancy model to another, some rules apply to all redundancy models:

- The overall goal of the Availability Management Framework is to keep as many active assignments as requested by the configuration for all service instances (which are administratively unlocked). If a service unit that is active for a service instance goes out-of-service, the Availability Management Framework automatically assigns the active HA state to a service unit that is already standby for the service instance if there is one.
- In the absence of administrative operations or error recovery actions being performed, only active and (possibly) standby HA states are assigned to the service units for particular service instances.

3.2.1.6 HA Readiness State of a Service Unit per Service Instance

The **HA readiness state of a service unit for a service instance** is the aggregation of the various HA readiness states of the components included in the service unit for the component service instances included in the service instance. This state reflects the ability of the service unit to assume the active or the standby assignments for the service instance. This state further qualifies the readiness state of a service unit with respect to each particular service instance protected by the service group. The Availability Management Framework can use this state and the readiness state to decide which service unit is most appropriate for an HA state assignment. For details about the HA readiness state of a component for a component service instance, refer to Section 3.2.2.5.

The HA readiness state of a service unit for a service instance can take the following values:

- **ready-for-assignment** - The Availability Management Framework sets this state to a service unit for a service instance if for each component service instance of the service instance there is at least one component in the service unit with an HA readiness state set to ready-for-assignment, so that the component service instance can be assigned to the component. The service unit can take assignments in any HA state for the service instance. If this value is set when the service unit is not assigned or assigned standby for the service instance, the Availability Management Framework must evaluate whether the service unit should be assigned the affected service instance or whether the standby assignment should be changed to an active assignment based on the overall status and redundancy model of the containing service group and also on whether the redundancy requirements of the service instance are being met.
- **ready-for-active-degraded** - The Availability Management Framework sets this state to a service unit for a service instance if
  - there is at least one component service instance in the service instance for which there is no component in the service unit with an HA readiness state set to ready-for-assignment, so that the component service instance can be assigned to the component, but there is at least one component in the service unit with an HA readiness state set to ready-for-active-degraded to which the component service instance can be assigned, and
  - for each other component service instance of the service instance, there is at at least one component in the service unit with an HA readiness state set to ready-for-assignment or ready-for-active-degraded, so that the component service instance can be assigned to the component.

This state indicates that the Availability Management Framework should avoid assigning to the service unit the active HA state for the service instance, because the service unit is not yet ready for it, and the active assignment would have a negative impact on the quality of the service being provided.

If this value is set when the service unit is assigned or being assigned active or quiescing for the service instance, the Availability Management Framework must attempt to reassign the service instance to another service unit whose HA readiness state is set to ready-for-assignment for this service instance. However, if this value is set when the service unit is assigned or being assigned standby or quiesced for the service instance, the assignment is not affected.

If this value is set when the service instance is not assigned to the service unit, the Availability Management Framework must evaluate whether the service unit should be assigned standby for the affected service instance based on the overall status and redundancy model of the containing service group and also on whether the redundancy requirements of the service instance are being met.

- **not-ready-for-active** - The Availability Management Framework sets this state to a service unit for a service instance if
  - there is at least one component service instance in the service instance for which there is no component in the service unit with an HA readiness state set to ready-for-assignment or ready-for-active-degraded, so that the component service instance can be assigned to the component, but there is at least one component in the service unit with an HA readiness state set to not-ready-for-active to which the component service instance can be assigned, and
  - for each other component service instance of the service instance, there is at at least one component in the service unit with an HA readiness state set to ready-for-assignment, ready-for-active-degraded, or not-ready-for-active, so that the component service instance can be assigned to the component.

  This state indicates that the Availability Management Framework must not assign to the service unit any of the HA states active or quiescing for a service instance.

  If this value is set when the service unit is assigned or being assigned active or quiescing for the service instance, the Availability Management Framework must remove the assignment or change it to standby. However, if this value is set when the service unit is assigned or being assigned standby or quiesced for the service instance, the assignment is not affected.

If this value is set when the service instance is not assigned to the service unit, the Availability Management Framework must evaluate whether the service unit should be assigned standby for the affected service instance based on the overall status and redundancy model of the containing service group and on whether the redundancy requirements of the service instance are being met.

- **not-ready-for-assignment** - The Availability Management Framework sets this state to a service unit for a service instance if, at least for one of the component service instance of the service instance, there is no component in the service unit with an HA readiness state set to ready-for-assignment, ready-for-active-degraded, or not-ready-for-active, so that the component service instance can be assigned to the component.
  This state indicates that the Availability Management Framework must not assign this service instance to the service unit.
  If this value is set when the service instance is assigned or being assigned to the service unit, the Availability Management Framework must remove the assignment.

3.2.2 Component States

The overall state of a component is a combination of a number of underlying states. A description of these underlying states is given in the next sections.

Note: No restriction exists in the applicability of various states of a component and their values described in the following subsections to proxied components. However, if the status of a proxied component changes to unproxied (typically, when its proxy component fails, and no proxy can be engaged to “proxy” the proxied component), the values for various states of this proxied component reflect the last know value of the corresponding states before its status became unproxied.

3.2.2.1 Presence State

The **presence state** of a component reflects the component life cycle. It takes one of the following values:

- uninstantiated
- instantiating
- instantiated
- terminating
- restarting
- instantiation-failed
- termination-failed

The presence state of a component is set to **instantiating** when the Availability Management Framework invokes

- the `saAmfProxiedComponentInstantiateCallback()` function (see Section 7.10.2), or
- the `saAmfContainedComponentInstantiateCallback()` function (see Section 7.10.4), or
- the `saAmfCSISetCallback()` function (see Section 7.9.2), or
- when it executes the `INSTANTIATE` CLC-CLI command (see Section 4.6),

as applicable according to Table 37 on page 439, to instantiate the component.

The presence state of a component is set to **instantiated** when the `INSTANTIATE` CLC-CLI command returns successfully (only for non-proxied, non-SA-aware components) or the component is registered successfully with the Availability Management Framework (for SA-aware or proxied components).

If, after all possible retries, a component cannot be instantiated, the presence state of the component is set to **instantiation-failed**.

The following actions set the presence state of a component to **terminating**:

- The Availability Management Framework invokes the
  - `SaAmfComponentTerminateCallbackT` function (see Section 7.10.1),
  - or the `saAmfCSIRemoveCallback()` function (see Section 7.9.3),
  - or it executes the `TERMINATE` CLC-CLI function (see Section 4.7),

  as applicable according to Table 37 on page 439, to terminate the component gracefully.
- The Availability Management Framework **abruptly** terminates the component by using one of the following interfaces, as applicable according to Table 37 on page 439:
  - by executing the `CLEANUP` CLC-CLI command (see Section 4.8),
  - or by invoking the `saAmfContainedComponentCleanupCallback()` (see Section 7.10.5),
  - or by invoking the `saAmfProxiedComponentCleanupCallback()` (see Section 7.10.3).

A component will enter the **uninstantiated** state if it is successfully terminated or cleaned up; it enters the **termination-failed** state if the cleanup operation fails.

If an **instantiated** component fails, the Availability Management Framework will make an attempt to restart the component, provided that restart is allowed for the component.

A component is restarted by the Availability Management Framework in the context of error recovery and repair actions (for details, see Section 3.11) or in the context of a restart administrative operation (for details, see Section 9.4.7). **Restarting** a component means first terminating it and then instantiating it again (see Section 3.11.1.2). Two different actions shall be undertaken by the Availability Management Framework regarding the component service instances assigned to a component when the component restart is needed:

- Keep the component service instances assigned to the component while the component is restarted. This action is typically performed when it is faster to restart the component than to reassign the component service instances to another component. In this case, the presence state of the component is set to restarting while the component is being terminated and until it is instantiated again (or a failure occurs). Internally, in this particular scenario, the Availability Management Framework withdraws and reassigns exactly the same HA state on behalf of all component service instances to the component as was assigned to the component for various component service instances before the restart procedure, without evaluating the various criteria that the Availability Management Framework would normally assess before making such an assignment.
- Reassign the component service instances currently assigned to the component to another component before terminating/instantiating the component. In this case, the presence state of the component is not set to restarting but transitions through the other presence state values (typically in the absence of failures: terminating, uninstantiated, instantiating, and then instantiated) as the component is terminated and instantiated again.

The choice between these two policies is based on the `saAmfCompDisableRestart` configuration attribute of each component (see the `SaAmfComp` object class in Section 8.13.2).

When a node leaves the cluster, the Availability Management Framework sets the presence state of all components included on that node to uninstantiated, except for components that are in the instantiation-failed or termination-failed state.

Table 5 shows the possible presence states of the components of a service unit for each valid presence state of the service unit:.

Table 5 Presence State of Components of a Service Unit

Service Unit	Included Components
uninstantiated	uninstantiated
instantiating	uninstantiated instantiating instantiated restarting
instantiated	instantiated restarting
terminating	terminating instantiated restarting uninstantiated
restarting	restarting
instantiation-failed	instantiation-failed uninstantiated instantiated terminating termination-failed
termination-failed	instantiated terminating termination-failed uninstantiated

3.2.2.2 Operational State

The **operational state** of a component reflects its error status. Valid values for the operational state of a component are:

- **enabled**: the Availability Management Framework is not aware of any error for this component, or a restart recovery action is in progress to recover from this error.
- **disabled**: the Availability Management Framework is aware of at least one error for this component that could not be recovered from by restarting the component or its service unit.

The described approach for operational state definition was chosen to reflect properly the capability of a component to be restarted within the time limits critical for the service it provides regardless the reason of the restart.

The Availability Management Framework becomes aware of an error for a component in the following circumstances:

- An error for the component is reported to the Availability Management Framework when the API function `saAmfComponentErrorReport_4()` is invoked. Such an error can be reported by the component itself, by another component, or by a monitoring facility (see `saAmfPmStart_3()`).
- The component fails to respond to the Availability Management Framework's healthcheck request or responds with an error.
- The component fails to initiate a component-invoked healthcheck in a timely manner.
- A command used by the Availability Management Framework to control the component life cycle returned an error or did not return in time.
- The component fails to respond in time to an Availability Management Framework's callback.
- The component responds to an Availability Management Framework's state change callback (`SaAmfCSISetCallbackT`) with an error other than `SA_AIS_ERR_NOT_READY`.
- If the component is SA-aware, and it does not register with the Availability Management Framework within the preconfigured time-period after its instantiation (see Section 4.6).
- The invocation of the `saAmfProxiedComponentInstantiateCallback()` function of a proxy or the invocation of the `saAmfContainedComponentInstantiateCallback()` function of the associated container component returns with an error.

- If an SA-aware component finalizes a handle returned by the `saAmfInitialize_4()` function when it still has some registered components associated to it (see Section 7.1.1).
- The component terminates unexpectedly.
- When a fail-over recovery operation performed at the level of the service unit or the node containing the service unit triggers an abrupt termination of the component (for the term abrupt termination, see Section 3.2.2.1). For more details about recovery operations, refer to Section 3.11.1.3.

A component is enabled when the node containing it joins the cluster for the first time. It is set to disabled when the Availability Management Framework performs a fail-over recovery action on the component as a consequence of the component becoming faulty, or if its presence state is set to instantiation-failed or termination-failed. It is again enabled after a successful repair. When a restart recovery action is performed on a component, it is considered as an instantaneous, combined recovery and repair action; therefore, the operational state of the component remains enabled in that case.

It is the Availability Management Framework that determines the value for the operational state. The operational state of a component is not directly exposed to components by the Availability Management Framework API.

3.2.2.3 Readiness State

The operational state of a component is combined with the readiness state of its service unit to obtain the **readiness state** of the component. This state indicates whether a component is available to take component service instance assignments, and it is used by the Availability Management Framework to decide whether a component is eligible to receive component service instance assignments.

The readiness state of a component is defined as follows:

- **out-of-service**: the readiness state of a component is out-of-service if its operational state is disabled, or the readiness state of the service unit containing it is out-of-service. When the readiness state of a component is out-of-service, no component service instance can be assigned to it.
- **in-service**: the readiness state of a component is in-service if its operational state is enabled, and the readiness state of the service unit containing it is in-service. When a component is in the in-service readiness state, it is eligible for component service instance assignments; however, it is possible that it has not yet been assigned any component service instance.

- **stopping**: the readiness state of a component is stopping if its operational state is enabled, and the readiness state of the service unit containing it is stopping. When the readiness state of a component is stopping, no component service instance can be assigned to it. The standby component service instance assignments are removed immediately, but active component service instances are not removed before the component indicates to the Availability Management Framework to do so.

The following table summarizes how the readiness state of a component is derived from the component's operational state and the enclosing service unit's readiness state.

Table 6 Component's Readiness State

Service Unit's Readiness State	Component's Operational State	Component's Readiness State
in-service	enabled	in-service
stopping	enabled	stopping
out-of-service	enabled	out-of-service
in-service	disabled	out-of-service
stopping	disabled	out-of-service
out-of-service	disabled	out-of-service

3.2.2.4 HA State of a Component per Component Service Instance

For each component service instance assigned to a component within a service unit, the Availability Management Framework assigns an **HA state** to the component on behalf of the component service instance.

When the Availability Management Framework assigns an HA state to a service unit for a particular service instance, the action is actually translated into a set of subactions on the components contained in the service unit. These subactions consist in assigning an HA state to these components for the individual component service instances contained in the service instance.

The HA state of a component for a particular component service instance takes one of the following values (identical to the HA state of a service unit for a particular service instance):

- **active**: the component is currently responsible for providing the service characterized by this component service instance.
- **standby**: the component acts as a standby for the service characterized by this component service instance.
- **quiescing**: the component that had previously an active HA state for this component service instance is in the process of quiescing its activity related to this service instance. In accordance with the semantics of the shutdown administrative operation, this quiescing is performed by rejecting new users of the service characterized by this component service instance while still providing the service to existing users until they all terminate using it. When no user is left for that service, the component indicates that fact to the Availability Management Framework, which transitions the HA state to quiesced. The quiescing HA state is assigned as a consequence of a shutdown administrative operation.
- **quiesced**: the component that had previously the active or quiescing HA state for this component service instance has now quiesced its activity related to this component service instance, and the Availability Management Framework can safely assign the active HA state for this component service instance to another component. The quiesced state is assigned in the context of switch-over situations (for the description of switch-over, refer to Section 3.3).

As the sub-actions involved to change the HA state of individual components of the service unit will not complete at the same time, the HA state of a service unit for a service instance and the HA state of individual components for the component service instances contained in that service instance may differ.
The following table describes the possible combinations. Note that the occurrence of the states active, standby, quiescing, and quiesced, in this order, in a row at the component or component service instance level (second column), determines the state in the same row at the service unit or service instance level (first column). So, if the state active appears in a row at the component or component service instance level, the state in the same row at the service unit or service instance level is active. If a row at the component or component service instance level shows no active but rather a standby state, the state of the same row at the service unit or service instance level is standby. The same applies similarly for the quiescing and quiesced states.

Table 7 HA State of Component/Component Service Instance

HA State of Service Unit/ Service Instance	HA State of Component/ Component Service Instance
active	active quiescing quiesced (not assigned)
active	active standby (not assigned)
quiescing	quiescing quiesced (not assigned)
quiesced	quiesced (not assigned)
standby	standby quiesced (not assigned)
(not assigned)	(not assigned)

The first two rows of the previous table are used to identify the two possible but mutually exclusive combinations of HA state of components when the HA state of the service unit is active. The second row is specific for a transition of the HA state of the service unit from standby to active.

For simplicity of expression, the term **active assignment of/for a component service instance** (or simply **active assignment** if the context makes it clear which component service instance is meant) is used to mean the assignment of the active HA state to a component for this component service instance. Similar terms are also used for the other HA states, such as **standby assignment**.

When the Availability Management Framework assigns the active HA state to a component on behalf of a component service instance, the component must start to provide the service that is characterized by that component service instance. If unable to provide the service characterized by the component service instance, the component must set its HA readiness state for that component service instance accordingly and

reject the active assignment (for details about the HA readiness state, refer to Section 3.2.2.5).

When the Availability Management Framework assigns the standby HA state to a component on behalf of a component service instance, the component must prepare itself for a quick and smooth transition into the active HA state for that component service instance if requested by the Availability Management Framework. How the standby component prepares itself for this transition is very dependent on its implementation and may involve, for example, actions such as sharing access to checkpointed data with the active component. If unable to assume the standby assignment for the component service instance, the component must set its HA readiness state for that component service instance accordingly and reject the standby assignment (for details about the HA readiness state, refer to Section 3.2.2.5).

In switch-over situations (see Section 3.3), when the Availability Management Framework assigns the quiesced HA state to a component on behalf of a component service instance, the component must, as quickly as possible, get the work related to that component service instance into such a state that the work can be transferred to another component with as minimal service disruption as possible. This may mean different things depending on the nature of the work and the implementation of the component. Typically, the component should not take in new work related to the component service instance. For example, if work related to the service instance is delivered in the form of messages sent to a specific message queue, the component should stop retrieving messages from that queue. Work which is related to that component service instance and which is already in progress inside the component, should be checkpointed, so that it can be completed later on by the component that will take over. If the component or the way it interacts with its clients does not support checkpointing of on-going work, either the work needs to be completed immediately or an indication needs to be returned to the client indicating that it should submit that work later. If the component maintains some state associated with the component service instance, that state needs to be made available to the component that will take over the activity. Depending on the implementation of the component, this may imply, for example, writing the state in persistent storage or in a checkpoint, or packing it in a message and sending it to a particular message queue.

As a consequence of a shutdown administrative operation (see Section 9.4.6 on page 380), when the Availability Management Framework assigns the quiescing HA state to a component on behalf of a component service instance, the component must reject attempts from new users to access the service characterized by the component service instance and only continue to service existing users. When all users have terminated using the service corresponding to that component service instance, the component must notify this termination to the Availability Man-

agement Framework by invoking the `saAmfCSIQuiescingComplete()` function. The invocation of the `saAmfCSIQuiescingComplete()` function implicitly transitions the HA state of the component from quiescing to quiesced for that component service instance.

The Availability Management Framework performs the following actions when it assigns the active HA state to a service unit for a particular service instance:

- It invokes the `saAmfCSISetCallback()` callback of all SA-aware components for the components themselves.
- It invokes the `saAmfCSISetCallback()` callback of their proxy components for all proxied components. If the proxied component is a non-pre-instantiable component and is not already instantiated, the proxy instantiates the proxied component as part of performing the component service instance assignment.
- It runs the `INSTANTIATE` command for non-proxied, non-SA-aware components.

The Availability Management Framework performs the following actions regarding components when it assigns to a service unit an HA state other than active for a particular service instance:

⇒ It invokes the `saAmfCSISetCallback()` callback of all SA-aware components for the components themselves.

⇒ For the special case of a container CSI of this particular service instance for which the HA state of the container component was active, the Availability Management Framework performs the following actions, before it invokes the `saAmfCSISetCallback()` callback to set the new HA state of the container component for the container CSI.

- For each associated contained component and for each of its component service instances that has the active HA state and needs to be quiesced, the Availability Management Framework sets the HA state of the associated contained component to quiescing, if the change of the HA state of the container CSI was caused by a shutdown administrative operation on a service unit or on any entity containing the service unit; otherwise, it is set to quiesced.
- The Availability Management Framework waits for each associated contained component to quiesce for its component service instances (if the setting of the HA state to quiescing or quiesced was necessary), then it removes all component service instances assigned to the contained component and terminates it.

⇒ It invokes the `saAmfCSISetCallback()` callback of their proxy components for all proxied components. The proxy component terminates its non-pre-instantiable proxied components as part of performing the component service instance assignment.

⇒ It runs the `TERMINATE` command for non-proxied, non-SA-aware components.

The Availability Management Framework performs the following actions regarding components when it removes a service instance assignment from a service unit:

⇒ It invokes the `saAmfCSIRemoveCallback()` callback of all SA-aware components for the components themselves.

⇒ For the special case of a container CSI of this particular service instance for which the HA state of the container component was active, the Availability Management Framework performs the following actions, before it invokes the `saAmfCSIRemoveCallback()` callback to remove the active HA state from the container component for the container CSI.

- For each associated contained component and for each of its component service instances that has the active HA state and needs to be quiesced, the Availability Management Framework sets the HA state of the associated contained component to quiesced.
- The Availability Management Framework waits for each associated contained component to quiesce for its component service instances (if the setting of the HA state to quiesced was necessary), then it removes all component service instances assigned to the contained component and terminates it.

⇒ It invokes the `saAmfCSIRemoveCallback()` callback of their proxy components for all proxied components. The proxy component terminates its non-pre-instantiable proxied components as part of removing the component service instance assignment.

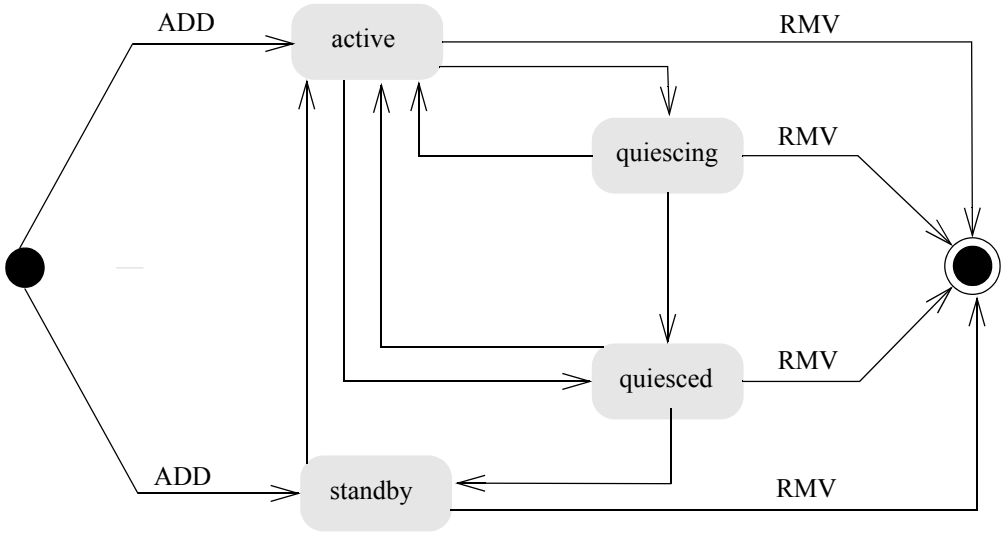
⇒ It runs the `TERMINATE` command for non-proxied, non-SA-aware components.

The instantiation of proxied, non-pre-instantiable components is performed by the proxy as part of the assignment of component service instances to the proxied component. Similarly, the termination of proxied, non-pre-instantiable components is performed by the proxy as part of the removal of component service instances from the proxied component. Hence, the Availability Management Framework never invokes the `saAmfProxiedComponentInstantiateCallback()` and `saAmfComponentTerminateCallback()` callback functions of the proxy for proxied, non-pre-instantiable components.

During an individual component restart induced by a fault encountered by the component, the component remains enabled. Its readiness state can change according to changes in its presence state (as described in Section 3.2.2.1), and it is the readiness state that determines the Availability Management Framework's actions regarding the CSI assignments to the component.

The state diagram for a component service instance is shown in FIGURE 3.

FIGURE 3 State Diagram of the HA State of an SA-Aware Component for a CSI



ADD Transitions: `saAmfCSISetCallback(SA_AMF_CSI_ADD_ONE)`

RMV Transitions: `saAmfCSIRemoveCallback()`,

`saAmfTerminateCallback()`, cleanup operation (see Table 37 in Appendix A)

Other Transitions: `saAmfCSISetCallback(SA_AMF_CSI_TARGET_*)`

Table 8 shows combinations of the readiness state and the HA state for pre-instantiable components for a component service instance. Only the HA state is exposed to application developers.

Table 8 Application Developer View for Pre-Instantiable Components

Component's Readiness State	Component's HA state for a Component Service Instance
in-service	active standby quiescing quiesced
stopping	standby quiescing quiesced
out-of-service	[no HA state]

Table 9 shows combinations of the readiness state and the HA state for non-pre-instantiable components for a component service instance. Only the HA state is exposed to application developers.

Table 9 Application Developer View for Non-Pre-Instantiable Components

Component's Readiness State	Component's HA state for a Component Service Instance
in-service	active or no HA state
out-of-service	[no HA state]

3.2.2.5 HA Readiness State of a Component for a Component Service Instance

The **HA readiness state of a component for a component service instance** is used to further qualify the ability of the component to be assigned the component service instance in a particular HA state.

The HA readiness state is not used to reflect a failure of the component, but rather to reflect situations in which a healthy component is not ready to assume a particular assignment for a component service instance, either because the component is not in an internal state required for the assignment, or because some resources on which the assignment depends are not available.

The HA readiness state of a component for a component service instance is set

- automatically by the Availability Management Framework in the situations described below, or
- explicitly by the registered process for a pre-instantiable component when the registered process invokes the `saAmfHAReadinessStateSet()` function (that is, this state cannot be modified for non-pre-instantiable components).

The HA readiness state of a component for a component service instance can take the following values:

- **ready-for-assignment** - This value indicates that the component is ready for an assignment of the component service instance in any HA state. This is the only value that the HA readiness state of a non-pre-instantiable component may take; this value is enforced by the Availability Management Framework, and it cannot be modified. The Availability Management Framework automatically sets the HA readiness state of pre-instantiable components to this value for all its potential component service instances when the presence state of the component becomes uninstantiated.
- **ready-for-active-degraded** - This value indicates that though the component is able to be assigned the active HA state for the component service instance, such an assignment would impact the quality of the service being provided and should only be issued when no other component has an HA readiness state set to ready-for-assignment for this component service instance, so that this other component could assume the active assignment. This value is typically used by a component that has the standby assignment for a component service instance but has not yet fully synchronized its internal state with the component assigned active, quiescing, or quiesced for the component service instance. The component is able to become active but may have to drop all client sessions whose state is not yet synchronized.
  This value can only be set for pre-instantiable components and is never set automatically by the Availability Management Framework.
- **not-ready-for-active** - This value indicates that the component cannot take an active or quiescing assignment for the component service instance, and standby or quiesced are the possible assignments that the component can assume for the component service instance. This value is typically used when the component has a missing dependency that prevents it from taking the active assignment for the component service instance. It may also be used in situations where the component has not synchronized its state with the component assigned active, quiescing, or quiesced for the component service instance, and it cannot take the active assignment with a unsynchronized state. If this value is set when the component is already assigned active or quiescing for the component service instance, the Availability Management Framework must either

remove the assignment or change it to standby.
This value can only be set for pre-instantiable components and is never set automatically by the Availability Management Framework.

- **not-ready-for-assignment** - This value indicates that the component is not able to take any assignment for the component service instance. This value is typically used if the component detects a missing dependency that prevents it from assuming any assignment for the component service instance. If this value is set when the component is already assigned for the component service instance, the Availability Management Framework must remove the assignment.
This value is can only be set for pre-instantiable components and is never set automatically by the Availability Management Framework.

A component does not typically know in advance which component service instance assignments it will receive from the Availability Management Framework. If a component receives a new component service instance assignment request (see `SaAmfCSISetCallbackT`) for which it is not ready, it must set its HA readiness state for this component service instance accordingly (by invoking the `saAmfHAReadinessStateSet()` function), before it responds to the assignment request with the `SA_AIS_ERR_NOT_READY` error (by invoking the `saAmfResponse_4()` function) to prevent the Availability Management Framework from treating the error as a component failure.

Valid combinations of component readiness, component HA readiness for a component service instance, and HA state for a component service instance are listed in Table 10 (the quiesced transitional state is not shown),

Table 10 Combinations of States for a Component

Readiness State	HA Readiness State for a CSI	HA State for a CSI
in-service	ready-for-assignment	all
	ready-for-active-degraded	all
	not-ready-for-active	no assignment, standby
	not-ready-for-assignment	no assignment
stopping	ready-for-assignment	no assignment, quiescing
	ready-for-active-degraded	no assignment, quiescing
	not-ready-for-active	no assignment
	not-ready-for-assignment	no assignment
out-of-service	all	no assignment

3.2.3 Service Instance States

3.2.3.1 Administrative State

The **administrative state** of a service instance is manipulated by the system administrator. Valid values for the administrative state of a service instance are:

- **unlocked**: HA states can be assigned to service units on behalf of the service instance.
- **locked**: no HA state can be assigned to service units on behalf of the service instance.
- **shutting-down**: the service instance is shutting down gracefully. This means that all assignments of all its component service instances are quiescing or quiesced assignments.

The administrative state of a service instance is not directly exposed to components by the Availability Management Framework API.

The administrative state of a service instance is persistent even when all nodes within the cluster are rebooted.

Note: The administrative state value of locked-instantiation is not a valid state value for a service instance, as a service instance cannot be terminated and made non-instantiable as other logical entities may be.

3.2.3.2 Assignment State

The **assignment state** of a service instance indicates whether the service represented by this service instance is being provided or not by some service unit. Valid values for the assignment state of a service instance are:

- **unassigned**: a service instance is said to be unassigned if no service unit has the active or quiescing HA state for this service instance.
- **fully-assigned**: a service instance is said to be fully-assigned if and only if
  - the number of service units having the active or quiescing HA state for the service instance is equal to the preferred number of active assignments for the service instance, which is defined in the redundancy model of the corresponding service group (see Section 3.6), and
  - the number of service units having the standby HA state for the service instance is equal to the preferred number of standby assignments for the service instance, which is defined in the redundancy model of the corresponding service group (see Section 3.6).
- **partially-assigned**: a configured service instance that is neither unassigned nor fully-assigned is said to be partially-assigned.

The following table shows the preferred number of active and standby assignments, for various redundancy models (additionally, refer to Section 3.6):

Table 11 Preferred Number of Active and Standby Assignments

Redundancy Model	Preferred Number of Active Assignments	Preferred Number of Standby Assignments
2N	1	1
N+M	1	1
N-Way	1	as configured for the service instance
N-Way Active	as configured for the service instance	0
No-Redundancy	1	0

It is the Availability Management Framework that determines the value of the assignment state.

The assignment state of a service instance is not directly exposed to components by the Availability Management Framework API.

When a service instance enters the unassigned state, an alarm will be issued. For other changes in the assignment state, appropriate notifications will be issued (see Chapter 11).

3.2.4 Component Service Instance States

The Availability Management Framework does not define any states for a component service instance; instead states are defined for the service instance to which this component service instance pertains.

3.2.5 Service Group States

The only state defined by the Availability Management Framework for service groups is the **administrative state**. It can be manipulated by the system administrators and is an extension of the administrative state proposed by the ITU X.731 state management model ([9]). Valid values for the administrative state of a service group are:

- **unlocked**: the service group has not been directly prohibited from providing service by the administrator.
- **locked**: the service group has been administratively prohibited from providing service.
- **locked-instantiation**: the administrator has prevented all service units of the service group from being instantiated by the Availability Management Framework.
- **shutting-down**: the administrator has prevented all service units contained within the service group from taking new service instance assignments and requested that existing service instance assignments be gracefully removed. When all service instances assigned to all the service units within the service group have finally been removed, the administrative state of the service group transitions to locked, that is, the administrative state of the service group is locked after completion of the shutting down operation.

The Availability Management Framework uses the administrative state of the service group to determine the readiness state of the service units of the service group, as described in Section 3.2.1.4.

The administrative state of a service group is persistent even when all nodes within the cluster are rebooted.

The administrative state of a service group is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state of the service unit has an impact on component service instance assignments.

Note: Though a service group has no associated HA state, this specification uses the term “assign a service instance” to a service group, meaning that the service instance is assigned to one or more service units of the service group.

3.2.6 Node States

3.2.6.1 Administrative State

The **administrative state** of a node is an extension of the administrative state proposed by the ITU X.731 state management model ([9]). The administrative state of a node can be set by the system administrator. Valid values for the administrative state of a node are:

- **unlocked**: the node has not been directly prohibited from providing service by the administrator.
- **locked**: the node has been administratively prohibited from providing service.
- **locked-instantiation**: the administrator has prevented all service units of the node from being instantiated by the Availability Management Framework. Thus, all service units within the node are not instantiable.
- **shutting-down**: the administrator has prevented all service units contained within the node from taking new service instance assignments and requested that existing service instance assignments be gracefully removed. When all service instances assigned to all the service units within the node have finally been removed, the administrative state of the node transitions to locked, that is, the administrative state of the node is locked after completion of the shutting down operation.

The Availability Management Framework uses the administrative state of the node to determine the readiness state of the service units of the node, as described in Section 3.2.1.4.

The administrative state of a node is persistent even when all nodes within the cluster are rebooted.

The administrative state of a node is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state of the service unit has an impact on component service instance assignments.

3.2.6.2 Operational State

The **operational state** of the node reflects its error status. Valid values for the operational state of a node are:

- **enabled**: the operational state transitions from disabled to enabled when a successful repair action has been performed on the node (see Section 3.11.1.4).
- **disabled**: the operational state of a node transitions to disabled if a component of the node has transitioned to the disabled state and the Availability Management Framework has taken a recovery action at the level of the entire node (node switch-over, fail-over, or failfast).

The operational state of a node is enabled when the node joins the cluster for the first time. It is set to disabled when the Availability Management Framework performs a node-level recovery action. After a successful repair, the operational state of the node is set again to enabled by the entity performing the repair (Availability Management Framework or other entity). An administrative operation is provided to clear the disabled state of a node, so that an entity different from Availability Management Framework may perform the repair and declare the node repaired.

The Availability Management Framework uses the operational state of the node to determine the readiness state of the service units of the node, as described in Section 3.2.1.4. The operational state of a node is valid even after a node left the membership, as it is used to provide the information if the node was healthy or had a failure when leaving. The following explains the state transitions in detail:

The operational state of a node is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state of the service unit has an impact on component service instance assignments.

If a node is enabled and in the locked-instantiation administrative state when it leaves the cluster membership, the node stays enabled until it joins the cluster again.

If a node is enabled and not in the locked-instantiation administrative state when it leaves the cluster membership, the node becomes disabled while it is out of the cluster and becomes enabled again when it rejoins the cluster.

If a disabled node with the automatic repair attribute (see Section 3.11.1.4) turned on unexpectedly leaves the cluster membership, the Availability Management Frame-

work should assess the state of the node when the node rejoins the cluster membership to ascertain if it needs to proceed with the planned repair action that was potentially interrupted when the node unexpectedly left the cluster membership.

If a disabled node with the automatic repair attribute turned off leaves the cluster membership, the operational state of the node (and of its contained entities) is not modified when the node joins the cluster again. Note that the operational state of the node may have been reenabled by an SA_AMF_ADMIN_REPAIR administrative operation before the node rejoined the cluster, in which case the node becomes enabled upon rejoining the cluster.

3.2.7 Application States

The only state defined by the Availability Management Framework for an application is the **administrative state**. It can be manipulated by the system administrator and is an extension of the administrative state proposed by the ITU X.731 state management model ([9]). Valid values for the administrative state of an application are:

- **unlocked**: the application has not been directly prohibited from providing service by the administrator.
- **locked**: the application has been administratively prohibited from providing service.
- **locked-instantiation**: the administrator has prevented all service units of the application from being instantiated by the Availability Management Framework.
- **shutting-down**: the administrator has prevented all service units contained within the application from taking new service instance assignments and requested that existing service instance assignments be gracefully removed. When all service instances assigned to all the service units within the application have finally been removed, the administrative state of the application transitions to locked, that is, the administrative state of the application is locked after completion of the shutting down operation.

The Availability Management Framework uses the administrative state of the application to determine the readiness state of the service units of the application, as described in Section 3.2.1.4.

The administrative state of an application is persistent, even when all nodes within the cluster are rebooted.

The administrative state of an application is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state of the service unit has an impact on component service instance assignments.

3.2.8 Cluster States

The only state defined by the Availability Management Framework for a cluster is the **administrative state**. It can be manipulated by the system administrator and is an extension of the administrative state proposed by the ITU X.731 state management model ([9]). Valid values for the administrative state of a cluster are:

- **unlocked**: the cluster has been administratively allowed to provide service.
- **locked**: the cluster has been administratively prohibited from providing service.
- **locked-instantiation**: the administrator has prevented all service units of the cluster from being instantiated by the Availability Management Framework. Thus, all service units within the cluster are not instantiable.
- **shutting-down**: the administrator has prevented all service units contained within the cluster from taking new service instance assignments and requested that existing service instance assignments be gracefully removed. When all service instances assigned to all the service units within the cluster have finally been removed, the administrative state of the cluster transitions to locked, that is, the administrative state of the cluster is locked after completion of the shutting down operation.

The Availability Management Framework uses the administrative state of the cluster to determine the readiness state of the service units of the cluster, as described in Section 3.2.1.4.

The administrative state of a cluster is persistent across the reboot of the cluster.

The administrative state of a cluster is not directly exposed to components by the Availability Management Framework, but rather only indirectly, as the readiness state of the service unit has an impact on component service instance assignments.

3.2.9 Summary of States Supported for the Logical Entities

Table 12 summarizes the states that the Availability Management Framework supports for the logical entities of the system model.

Table 12 Summary of States Supported for the Logical Entities

Logical Entity	States
cluster	administrative
application	administrative
service group	administrative
node	administrative, operational
service unit	administrative, operational, readiness, HA readiness, HA, presence
component	operational, readiness, HA readiness, HA, presence
service instance	administrative, assignment
component service instance	-

The administrative states of service units, service groups, service instances, nodes, applications, and the cluster are completely independent in the sense that one does not affect the other. As an example, a service unit might be administratively unlocked while its enclosing node is locked. Whether the service unit is actually administratively prevented from providing service or not depends on the administrative state of the service unit and on the administrative states of its containing node, service group, application, and the cluster. The corresponding rules are given in Section 3.2.1.2.

Note that the administrative, presence, and operational states of a particular entity typically do not have a direct impact on each other. However, certain incidents may change more than one of these states, as explained next:

- A service unit failure can lead to its presence state changing to uninstantiated and its operational state changing to disabled. This incident is an example of an event that changes both the operational and presence states.
- When a service unit is administratively locked for instantiation (refer to Section 9.4.4 on page 375), its presence state changes to uninstantiated, and its administrative state changes to locked-instantiation, but its operational state remains unchanged.

Note that the states of service units and of their components are closely related and depend on each other:

- The operational state of a service unit is directly affected when the operational state of one of its components changes.
- The presence state of a service unit is directly affected when the presence state of one of its components changes.
- The readiness state of a component is affected when the readiness state of the containing service unit changes. Thus, administrative changes on a service unit have a direct impact on all components of a service unit.

In a similar way, the administrative state of the cluster, applications, AMF nodes, and service groups affect the readiness state of the contained service units and components.

Administrative commands are propagated from a parent to all its children by the readiness state. Dynamic changes are propagated in the opposite direction, from the component to the containing service unit, based on the component's presence or operational state.

3.3 Fail-Over and Switch-Over

The terms **component service instance fail-over** and **component service instance switch-over** are used to designate two scenarios in which a component assigned the active HA state for a particular component service instance loses the active assignment.

The term fail-over is used to designate a recovery procedure performed by the Availability Management Framework when a component with the active HA state for a component service instance fails (when, for instance, its operational state becomes disabled), and the Availability Management Framework decides to reassign the active HA state for the component service instance to another component. In a fail-over situation, the faulty component is abruptly terminated either by the fault itself (for example, a node failure) or by the Availability Management Framework, which promptly isolates the fault by executing the appropriate cleanup operation for the component (see Table 37 in Appendix A).

The term switch-over is used to designate circumstances in which the Availability Management Framework moves the active HA state assignment of a particular component service instance from one component `C1` to another component `C2` while the component `C1` is still healthy and capable of providing the service (that is, `C1` operational state is enabled). Switch-over operations are usually the consequence of administrative operations (such as lock of a service unit) or escalation of recovery procedures (for details about recovery escalations, see Section 3.11). To minimize the impact of the switch-over operation, the Availability Management Framework performs an orderly transition of the HA state of `C1` from active to quiesced before assigning the active HA state to `C2`. After `C2` has been assigned active for the component service instance, the Availability Management Framework will typically remove the component service instance assignment from `C1`.

Component service instance fail-overs and component service instance switch-overs are performed in the context of service instance fail-overs and service instance switch-overs. The terms **service instance fail-over** and **service instance switch-over** are used to designate situations in which the Availability Management Framework removes the active HA state of a service unit for a particular service instance. A service instance switch-over operation is the consequence of an administrative operation such as a lock operation, whereas a fail-over operation is the consequence of a failure recovery. As described in Section 3.11, it should be noted that depending on configuration options, a service instance fail-over may be implemented as a fail-over of all component service instances assigned to the failed component, while component service instances assigned to non faulty components are simply switched-over.

When the active assignment of a service instance to a service unit is removed from the service unit in the context of a fail-over or a switch-over, the Availability Management Framework must choose another service unit to take that active assignment. All other service units in the service group are considered following a decreasing preference order (i.e. most preferred service unit is considered first). The following rules are used to order the service units:

- Service units that are assigned standby for the service instance are preferred over other service units. This rule applies only to the 2N, N+M, and N-way redundancy models (for a description of redundancy models, refer to Section 3.6).
- If several service units are assigned standby for the service instance, a service unit with a higher ranking standby assignment for that service instance is preferred over a service unit with a lower ranking standby assignment. This rule applies only to the N-way redundancy model (see Section 3.6.4).
- Instantiated service units are preferred over uninstantiated service units.

The first (or possibly the only one) service unit in this ordered list that can take the active assignment for the service instance without exceeding the resource capacity of its AMF node (see Section 3.6.1.3) is chosen by the Availability Management Framework.

3.4 Possible Combinations of States for Service Units

3.4.1 Combined States for Pre-Instantiable Service Units

Table 14 and FIGURE 4 show the possible combinations of states for pre-instantiable service units. The first column of Table 14 contains the combined administrative states of the service unit and of the entities that enclose it, that is, service group, node (for a local service unit), application, and cluster. The combined administrative state has been introduced only for convenience, and its respective values are explained in Table 13.

Table 13 Combined Administrative States for the Service Unit

Combined Administrative State	Meaning
“unlocked”	The service unit and all its enclosing entities are in the unlocked administrative state.
“locked”	• One or more of the entities service unit and its enclosing entities are in the locked administrative state and • neither the service unit nor any of its enclosing entities are in the locked-instantiation administrative state.
"locked-instantiation"	• One or more of the entities service unit and its enclosing entities are in the locked-instantiation administrative state and • the service unit and all its enclosing entities that are not in the locked-instantiation state are either in the unlocked or locked administrative states.
"shutting-down"	• One or more of the entities service unit and its enclosing entities are in the shutting-down administrative state and • the service unit and all enclosing entities that are not in the shutting-down state are in the unlocked administrative state.

The terms “Operational”, “Presence”, “Readiness”, and “HA State” in the heading of Table 14 refer to the respective states of a service unit. The operational state of the node hosting the service unit is not shown in this table, but its effect is as follows: unless otherwise stated in footnotes to table rows, all rows in the table apply if the operational state of the node is enabled. If its operational state is disabled, only the rows containing “disabled” in the second column apply, irrespective of whether the operational state of the service unit is enabled or disabled.

Table 14 Combined States for Pre-Instantiable Service Units

Combined Administrative State from Table 13	Operational	Presence	Readiness	HA
“locked”	enabled	uninstantiated instantiating instantiated restarting	out-of-service	[no HA state]
“locked”	disabled	uninstantiated instantiation-failed terminating termination-failed	out-of-service	[no HA state]
“unlocked”	enabled	instantiated restarting	in-service	any
“unlocked”	enabled	uninstantiated instantiating terminating	out-of-service	[no HA state]
“unlocked”	disabled	uninstantiated instantiation-failed terminating termination-failed	out-of-service	[no HA state]
“shutting-down”	enabled	instantiated restarting	stopping	quiescing quiesced
“shutting-down”	enabled	instantiating terminating	stopping	[no HA state]
“shutting-down”	enabled	uninstantiated	out-of-service	[no HA state]

Table 14 Combined States for Pre-Instantiable Service Units (Continued)

Combined Administrative State from Table 13	Operational	Presence	Readiness	HA
“shutting-down”	disabled	uninstantiated instantiation-failed instantiating instantiated[1] restarting terminating termination-failed	out-of-service	[no HA state]
“locked-instantiation”	enabled	uninstantiated terminating	out-of-service	[no HA state]
“locked-instantiation”	disabled	uninstantiated instantiation-failed terminating termination-failed	out-of-service	[no HA state]

1. This combination of states applies only if the node hosting the service unit is disabled while the service unit itself is still enabled.

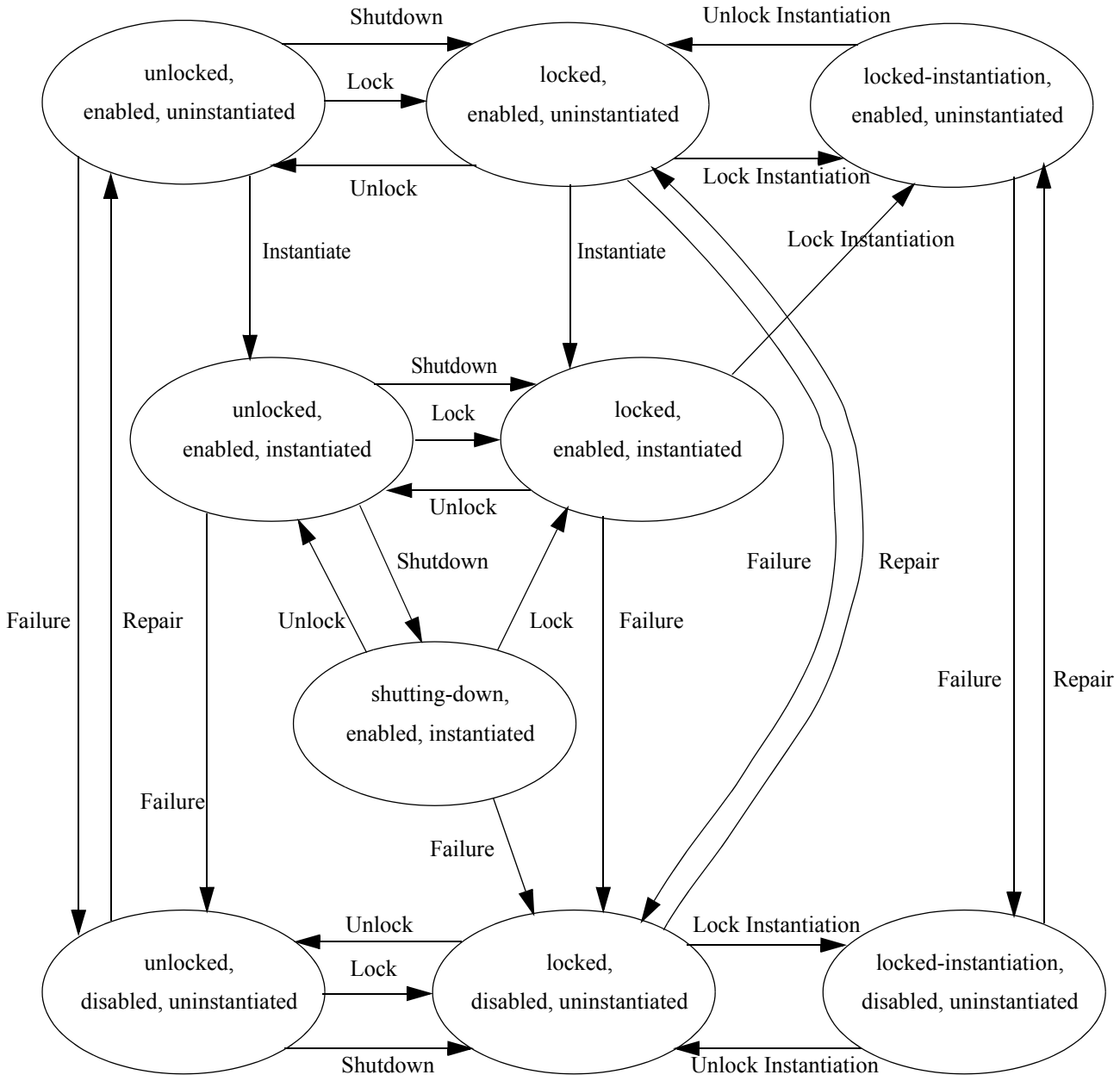
Reasons for a service unit to move from one combination of states to another:

- lock, lock-instantiation, shutdown, unlock-instantiation or unlock administrative operation,
- failure of a component contained in the service unit, and which escalates to disabling the containing service unit, and thus leading to a clean up and uninstantiation of the service unit,
- repair of a failed service unit (by restarting the service unit, by rebooting the node, by invoking the `saAmfComponentErrorClear_4()` function, or by executing the `SA_AMF_ADMIN_REPAIRED` administrative operation), and
- all components contained in the service unit leaving the `SA_AMF_HA_QUIESCING` state for all their component service instances (labeled with "Stopped" in the next diagram);
- concerning service units containing contained components, the following additional reasons: termination or restart of the associated container component, administrative lock or shutdown of a service instance containing the corresponding container CSI, or administrative lock or shutdown of the service unit containing the associated container component.

Some of the important state transitions for a pre-instantiable service unit are shown in FIGURE 4. The following simplifications were made in this figure:

- only the presence state instantiated and uninstantiated are considered;
- the states shown as locked, unlocked, and so on refer to the administrative state of the service unit; it is assumed that the enclosing entities are all in the unlocked administrative state;
- for the transitions among states, only “Instantiate” and administrative operations are considered (Lock, Unlock, and so on), and they apply only to the service unit;
- the operational state of the node hosting the service unit is enabled.

FIGURE 4 State Transitions for Pre-Instantiable Service Units



3.4.2 Combined States for Non-Pre-Instantiable Service Units

Table 15 and FIGURE 5 show the possible combinations of states for non-pre-instantiable service units. The terms “Operational”, “Presence”, “Readiness”, and “HA State” in the heading of Table 15 refer to the respective states of a service unit. The operational state of the node hosting the service unit is not shown in this table, but its effect is as follows:

All rows in the table apply if the operational state of the node is enabled. If its operational state is disabled, only the rows containing “disabled” in the second column apply, irrespective of whether the operational state of the service unit is enabled or disabled.

Table 15 Combined States for Non-Pre-Instantiable Service Units

Combined Administrative State from Table 13	Operational	Presence	Readiness	HA
“locked”	enabled	uninstantiated instantiating restarting terminating	out-of-service	[no HA state]
“locked”	disabled	uninstantiated instantiation failed terminating termination failed	out-of-service	[no HA state]
“unlocked”	enabled	uninstantiated	in-service	[no HA state]
“unlocked”	enabled	instantiating instantiated restarting	in-service	active
“unlocked”	disabled	uninstantiated instantiation failed terminating termination failed	out-of-service	[no HA state]
“shutting-down”	enabled	instantiating instantiated restarting	stopping	quiescing
“shutting-down”	enabled	uninstantiated terminating	stopping	[no HA state]
“shutting-down”	disabled	uninstantiated instantiation-failed terminating termination-failed	out-of-service	[no HA state]
“locked-instantiation”	enabled	uninstantiated	out-of-service	[no HA state]
“locked-instantiation”	disabled	uninstantiated instantiation failed terminating termination failed	out-of-service	[no HA state]

Reasons for a service unit to move from one combination of states to another:

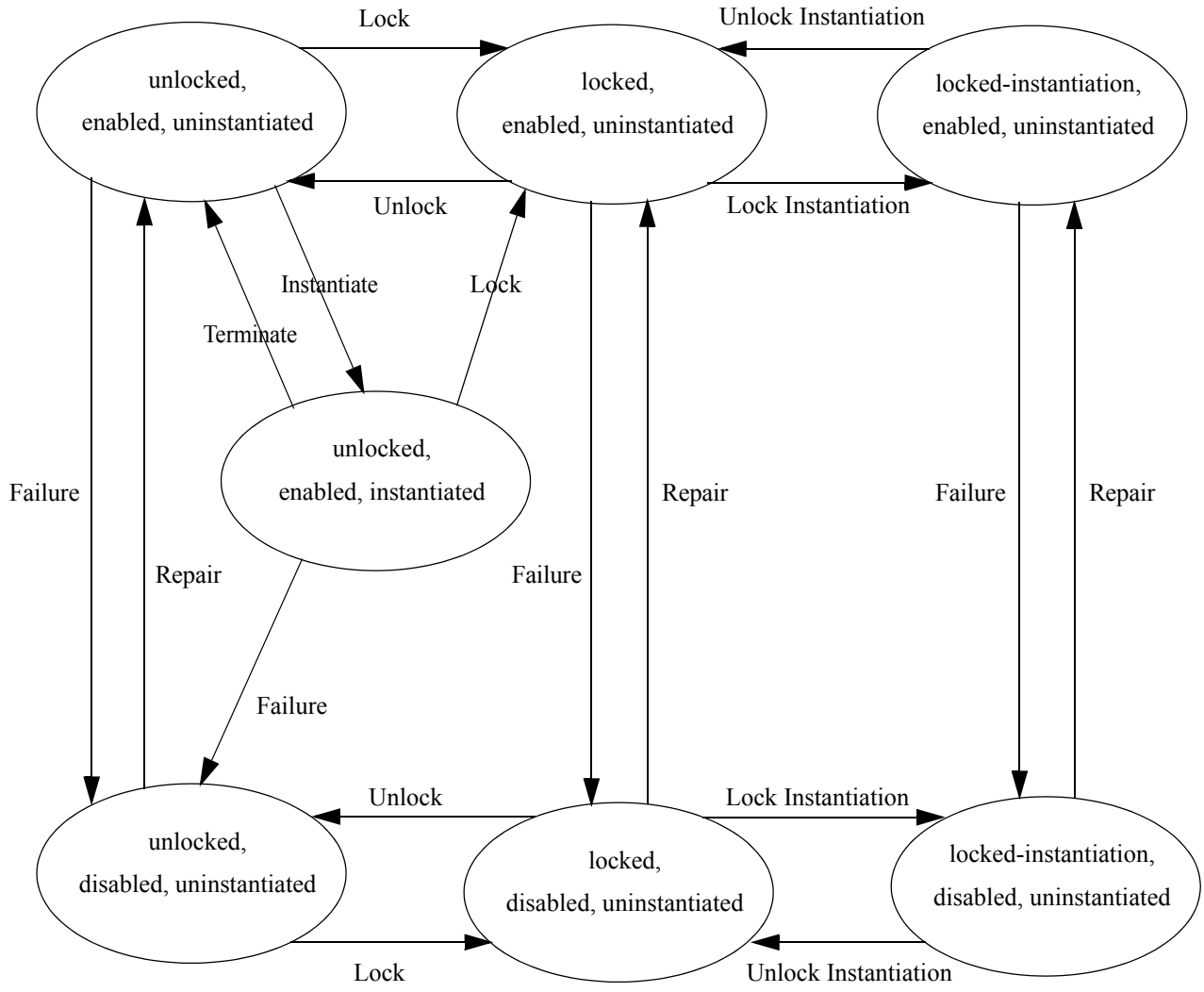
- lock, lock-instantiation, shutdown, unlock-instantiation, or unlock administrative operation,
- failure of a component contained in the service unit, which escalates to disabling the containing service unit, and thus cleaning up and uninstantiating the service unit,
- service unit uninstantiated by the Availability Management Framework, and
- service unit instantiated by the Availability Management Framework.

Some of the important state transitions for a non-pre-instantiable component are shown in FIGURE 5.

The following simplifications were made in this figure:

- only the presence state instantiated and uninstantiated are considered;
- the states shown as locked, unlocked, and so on refer to the administrative state of the service unit; it is assumed that the enclosing entities are all in the unlocked administrative state;
- for the transitions among states, only “Instantiate”, “Terminate”, and administrative operations are considered (Lock, Unlock, and so on), and they apply only to the service unit;
- the operational state of the node hosting the service unit is enabled;
- no transitions and states induced by the shutdown administrative operation are shown.

FIGURE 5 State Transitions for Non-Pre-Instantiable Service Units



3.5 Component Capability Model

To accommodate possible simplifications in component development, whereby components may implement only restricted capabilities, the Availability Management Framework defines the **component capability model**. The component capability always applies to a component for a component service type. Note that the letters x and y in the name of a component capability indicate only multiplicity of numbers of active or standby component service instances of a certain component service type. The precise values of the maximum number of active and standby CSIs are configured by setting the `saAmfCompNumMaxActiveCSIs` and `saAmfCompNumMaxStandbyCSIs` configuration attributes of the `SaAmfCompCsType` association class (see Section 8.13.2).

- **x_active_and_y_standby**: for a certain component service type, the component supports all values of the HA state, and it can have the active HA state for x component service instances and the standby HA state for y component service instances at a time.
- **x_active_or_y_standby**: for a certain component service type, the component supports all values of the HA state. It can be assigned either the active HA state for x component service instances or the standby HA state for y component service instances at a time.
- **1_active_or_y_standby**: for a certain component service type, the component supports all values of the HA state. It can be assigned either the active HA state for only one component service instance or the standby HA state for y component service instances at a time.
- **1_active_or_1_standby**: for a certain component service type, the component supports all values of the HA state, and it can be assigned either the active HA state or the standby HA state for only one component service instance at a time.
- **x_active**: for a certain component service type, the component cannot be assigned the standby HA state for component service instances, but it can be assigned the active HA state for x component service instances at a time.
- **1_active**: for a certain component service type, the component cannot be assigned the standby HA state for component service instances, but it can be assigned the active HA state for only one component service instance at a time.
- **non-pre-instantiable**: for a certain component service type, the component provides service as soon as it is started. The Availability Management Framework delays the instantiation of the component to the time when the component is assigned the active HA state on behalf of a component service instance. When the active HA state for a component service instance is removed from the component, the Availability Management Framework terminates the component. Such a component is termed non-pre-instantiable.

Service units may hold components supporting different capability models. The number of service instances assigned to a service unit depends on the number of component service instances supported by the components included in the service unit per component service type.

3.6 Service Group Redundancy Model

By configuration, a service group has associated with it a **service group redundancy model** (specified by the `saAmfSgtRedundancyModel` configuration attribute of the `SaAmfSGType` object class, shown in Section 8.9). The service units within a service group provide service availability to the service instances that they support according to the particular service group redundancy model.

The redundancy models are described in this chapter in terms of the rules followed by the Availability Management Framework when it assigns the active and standby HA state to service units of a service group for one or several service instances.

The assignment of the quiesced and quiescing HA states to the service units for particular service instances is not described here, as these states are not an integral part of the redundancy models definition, but rather transition states used by the Availability Management Framework to perform switch-over operations or implement the shutdown administrative operation.

In the remainder of this description, a service unit assigned the quiescing HA state for a service instance must be accounted as if the same service unit were assigned the active HA state for that same service instance.

The transient quiesced HA state is peculiar, as during a switch-over operation, a service unit assigned the quiesced HA state for a service instance must be accounted as being assigned either the active or standby HA state for that service instance.
In the remainder of this description, a service unit assigned the quiesced HA state for a particular service instance should be accounted as a service unit assigned the active HA state for that service instance if no other service unit has the active HA state assigned for that service instance; otherwise, it should be accounted as a service unit assigned the standby HA state for that service instance.

In the remainder of this description, a service unit that has the stopping readiness state must be accounted as an in-service service unit.

This specification defines the following service group redundancy models:

- 2N
- N+M
- N-way
- N-way active
- no-redundancy

These service group redundancy models are not exposed in the APIs of this specification. Note that the N in the 2N model refers to the number of service groups, whereas N and M, when used in the other models, refer to service units. This usage of N in the 2N model is due to common usage of the term 2N to refer to 1+1 active/standby redundancy configurations, which can be repeated N times.

Each redundancy model and the common characteristics of all or most of the redundancy models are explained in the following sections. Section 3.6.7 on page 182 describes the effect of administrative operations on the redundancy models.

3.6.1 Common Characteristics

Note: The following description uses several ordered lists like ordered list of service units or ordered list of service instances. The order of the elements in the list is based on the relative importance of these elements. The terms **rank** or **ranking** are used as synonyms to this order. Similarly, **ranked list** is also used as a synonym to ordered list.

3.6.1.1 Common Definitions

The following definitions and concepts are common to all the supported redundancy models.

- **Instantiable service units**: a service unit is instantiable if and only if all the following conditions are met:
  - ⇒ it is configured in the Availability Management Framework;
  - ⇒ it is contained in a CLM node that is currently a member of the cluster;
  - ⇒ it is contained in an AMF node whose operational state is enabled;
  - ⇒ its presence state is uninstantiated;
  - ⇒ its operational state is enabled;
  - ⇒ none of the relevant entities (service group, AMF node, etc.) has the administrative state locked-instantiation (however, their administrative state can be locked).

  Note that pre-instantiable service units in the instantiable set are out-of-service.
- **in-service service units**: these are the service units that have a readiness state of either in-service or stopping.

- **Instantiated service units**: in the context of this discussion, these are service units with the presence state of either instantiated, instantiating, or restarting. When the Availability Management Framework intends to select service units to be in the "instantiated service units" list, it chooses these service units from the instantiable service units that are not administratively locked at any of the levels service unit, containing node, service group, application, and the cluster. This selection is done according to the service unit rank defined for the particular redundancy models. The notion of “preferred number of in-service service units” is defined later for each redundancy model. See, for instance, Section 3.6.2.2. Note that the instantiable and instantiated sets are disjoint.
- **Assigned service units**: these are the service units that have at least one SI assigned to them. At runtime, this number is the value of the `saAmfSGNumCurrAssignedSUs` runtime attribute of the `saAmfSG` object class (see Section 8.9). If the Availability Management Framework needs to choose a service unit for assignment from the list of instantiated service units, it has to choose from the in-service instantiated service units.
- **Instantiated and non-instantiated spare service units**: all instantiated but unassigned service units are called **instantiated spare service units**, or simply spare service units. All non-instantiated service units of a service group are called **non-instantiated spare service units**. At runtime, these numbers of spare service units are the values of the `saAmfSGNumCurrInstantiatedSpareSUs` and `saAmfSGNumCurrNonInstantiatedSpareSUs` runtime attributes of the `saAmfSG` object class (see Section 8.9).
- **Ordered list of service units for a service group**: for each service group, an ordered list of service units defines the rank of the service unit within the service group. This rank is configured by setting the `saAmfSURank` attribute of the `saAmfSU` object class (see Section 8.10). The rank is represented by a positive integer. The lower the integer value, the higher the rank. The size of the list is equal to the number of service units configured for the service group. This ordered list is used to specify the order in which service units are selected to be instantiated. This list can also be used to determine the order in which a service unit is selected for SI assignments when no other configuration parameter defines it. It is possible that this list has only one service unit. However, to maintain the availability of the service provided by the service group, the list should include at least two service units.
  Default value: no default, the order is implementation-dependent.

- **Reduction Procedure**: the configuration of a service group describes the desired SI assignments that the Availability Management Framework should maintain when the preferred number of its service units can actually be instantiated; however, if one ore several service units fail to instantiate or are administratively taken out of service, all desired SI assignments may not be maintained anymore, and the reduction procedure describes for most redundancy models the behavior of the Availability Management Framework in such a situation.
- **No spare HA state**: as spare service units have no SI assigned to them, no "spare" HA state is defined for service units and components on behalf of service instance and component service instances, respectively. Hence, protection groups do not contain components of the spare service units, and so no changes need to be tracked for these components.
- **Auto-adjust option**: this option indicates that it is required that the SI assignments to the service units in the service group are transferred back to the most preferred SI assignments in which the highest-ranked available service units are assigned the active or standby HA states for those SIs. The auto-adjust option is configured by setting the `saAmfSGAutoAdjust` attribute of the `saAmfSG` object class (see Section 8.9). If the auto-adjust option is not set, the HA assignments to service units are kept unchanged even when a higher-ranked service unit becomes eligible to take assignments (for example, when a new node joins the cluster). For details when the auto-adjust option is initiated, refer to Section 3.6.1.2.

The following definitions are used in most, but not all, of the supported redundancy models.

- **Multiple (ranked) standby assignments**: for some redundancy models, it is possible that multiple service units are assigned the standby HA state for a given SI. These service units are termed the standby service units for this given SI. The standby service units are ranked, meaning that one service unit will be considered standby #1, another one standby #2, and so on. The rank is represented by a positive integer. The lower the integer value, the higher the rank. The standby service unit with the highest rank will be assigned the active HA state for a given service instance if the service unit that is currently active for that service instance fails. The rank of a standby service unit for an SI is configured by setting the `saAmfRank` attribute of a service unit identified by `safRankedSu` in the `SaAmfSIRankedSU` association class (see Section 8.11).
  When the Availability Management Framework assigns component service instances to a component, it notifies the component about the rank of its standby assignment. This additional information can be used for the component in preparing itself for the standby role.

- **Ordered list of SIs**: this ordered list is used to rank the SIs based on their importance. The rank of an SI is configured by setting the `saAmfSIRank` attribute of the `saAmfSI` object class (see Section 8.11). The rank is represented by a positive integer. The lower the integer value, the higher the rank. The Availability Management Framework uses this ranking to choose SIs to either support with less than the wanted redundancy or to drop them completely if the set of instantiated service units does not allow full support of all SIs. Note that the rank of an SI is global to the cluster and represents the importance of a SI in the whole cluster.
- **Redundancy level of a Service Instance**: the redundancy level is the number of service units being assigned an HA state for this service instance.

Though most redundancy models are applicable to service groups containing non-pre-instantiable service units (see Table 16 in Section 3.7), the description provided in the following sections only applies to service groups with pre-instantiable service units, as they lead to more complex situations. The behavior of the various redundancy models for service groups with non-pre-instantiable service units can be deduced from the following descriptions by taking into account the following restrictions for service groups with non-pre-instantiable service units:

- there are no spare service units and
- no standby service units;
- there is one and only one SI assignment per in-service service unit;
- the three sets of instantiated service units, in-service service units, and active service units are identical.

3.6.1.2 Initiation of the Auto-Adjust Procedure for a Service Group

If a service group is configured with the auto-adjust option set, that is, the `saAmfSGAutoAdjust` configuration attribute is set to `SA_TRUE` (see the `SaAmfSG` object class in Section 8.9), the Availability Management Framework should attempt to return the assignments of the service group back to the most preferred assignments (as defined in Section 3.6.1.1) as soon as possible. In general, auto-adjustment for a service group is needed in the following cases:

- A service unit configured for the service group becomes instantiable.
- The readiness state of a service unit configured for the service group becomes in-service.
- The HA readiness state of a service unit for a service instance changes in a way that allows the assignment of the service instance to the service unit.
- A locked service instance configured for the service group becomes unlocked.

- Free capacity of a node hosting a service unit becomes available such that a service instance can be assigned to a service unit.

When a service group becomes eligible for auto-adjustment, the Availability Management Framework can initiate the auto-adjust procedure for that service group immediately. This seems practical when an administrative action has made the service group eligible for the auto-adjust (for example, when a service instance is unlocked by the administrative operation). However, if the completion of a recovery/repair operation has made the service group eligible for auto-adjustment (for example, if a node joins the cluster after the repair), it is not so wise to run the auto-adjust procedure for the service group involving the newly repaired service units immediately. Thus, the service group-level configuration attribute **auto-adjust probation period** has been introduced (actually, the `saAmfSGAutoAdjustProb` configuration attribute in the `SaAmfSG` object class, shown in Section 8.9). When a service unit becomes available for auto-adjustment after a repair/recovery operation, the service unit enters its auto-adjust probation period, and it cannot thereby be used for auto-adjustment during this probation period. Note that the service group can be auto-adjusted using other service units, but auto-adjustment cannot use the service units in their auto-adjust probation periods. Also, the service unit on probation can and should be used in other operations such as switch-over and fail-over.

As soon as the auto-adjust probation period of a service unit elapses, the Availability Management Framework initiates the auto-adjust procedure for the corresponding service group.

By configuring the auto-adjust probation period appropriately, the administrator can ensure that the Availability Management Framework does not run into unwanted situations such as toggling the active service units due to, for example, intermittent failures of a service unit or inadequate repair operations.

3.6.1.3 AMF Node Capacity Limitation

In an AMF cluster, the capacity of AMF nodes in terms of resources like memory or computing power may vary. Some AMF nodes may have higher capacity in terms of these different resources than other AMF nodes. Also, different service instances may use these resources differently, one may require more memory, others may require more computing power. The usage of resources may also depend on the HA state the service instances are assigned to a service unit. Accordingly, service units of the same AMF node may be able to support more service instances of one service type than of another service type, and different AMF nodes may be able to support different combinations of service instances.

The Availability Management Framework configuration provides means for specifying the requirements of SIs in terms of logical resources and the capacity of nodes in terms of these resources. These logical resources can refer to physical resources on the mapped CLM node.

The configuration of an SI can specify a pair of **SI weights** for each of these resources, one weight when the SI is assigned active, quiescing, or quiesced to a service unit and the other when the SI is assigned standby to a service unit. The SI weight of an SI characterizes the load in term of these resources that the SI will impose on the node when it is assigned to a service unit. The respective configuration attributes belong to the `SaAmfSI` object class, shown in Section 8.11, and they are termed `saAmfSIActiveWeight` and `saAmfSIStandbyWeight`.

Each value of such a multi-value attribute has the following format:

```
<resource name>=<weight value>
```

For example:

```
{"resource_type_name1=weight_value1",
 "resource_type_name2=weight_value2",
                ...
 "resource_type_nameN=weight_valueN"}
```

Whereby for X = 1, 2, …, N, `resource_type_nameX` denotes a particular resource and `weight_valueX` the free capacity required of that resource to accommodate the assignment.

If no weight is specified for an active or standby assignment, or it is not specified for a particular type of resource, it means that the SI requires no resource at all or no resource for that type, respectively.

The **capacity** of an AMF node is specified in the `saAmfNodeCapacity` configuration attribute of the `SaAmfNode` object class, shown in Section 8.7.

Each value of such a multi-value attribute has the following format:

```
<resource name>=<capacity value>
```

For example:

```
{"resource_type_name1=capacity_value1",
 "resource_type_name2=capacity_value2",
                ...
 "resource_type_nameN=capacity_valueN"}
```

Whereby for X = 1, 2, ..., N, `resource_type_nameX` denotes a particular resource and `capacity_valueX` the capacity of that resource.

If no capacity is specified for a node, or it is not specified for a particular type of resource, it means that the capacity of the node for all resources or for the resource of that type, respectively, is not limited.

The Availability Management Framework applies the next rules when it performs an HA state assignment for any SI to any service unit on an AMF node for which capacities have been defined:

⇒ For each resource on an AMF node, the Availability Management Framework checks whether the sum of all SI weights of service instances assigned to service units contained in the AMF node does not exceed the capacity of the AMF node. An SI weight and an AMF node capacity refer to the same resource type if they both have the same name, for instance, `node_resourceY`. Note that each `weight_valueX` and its respective `capacity_valueX` must be expressed in the same units.

⇒ If at a given time the remaining capacity of the nodes that contain the service units configured for a service group is not sufficient to satisfy the assignments of the SIs protected by the service group, the Availability Management Framework must drop some SI assignments. This reduction is not limited to the assignments of the SIs of this particular service group, but rather interpreted cluster-wide. This reduction takes into account the global ranking of the service instances and the common set of nodes on which the different SIs can be assigned.
When calculating which SI assignment needs to be dropped first, the following rules should be observed:

- Standby assignments are dropped before any active assignment is dropped, starting from SIs of lower rank.
- Dropping of an active assignment—starting with the lower rank—is considered only when the dropping of standby assignments cannot free up the capacity required for the active assignment.

Note that as no SIs are assigned to spare service units, these service units are ignored when the load imposed on the AMF nodes on which they reside is computed.

Note: In the examples provided in the description of redundancy models, starting with Section 3.6.2, it is assumed that the HA readiness state of all service units for all service instances is 'ready-for-assignment' and that the hosting node resource capacity limits are not exceeded by any of the example assignments.

3.6.1.3.1 Examples

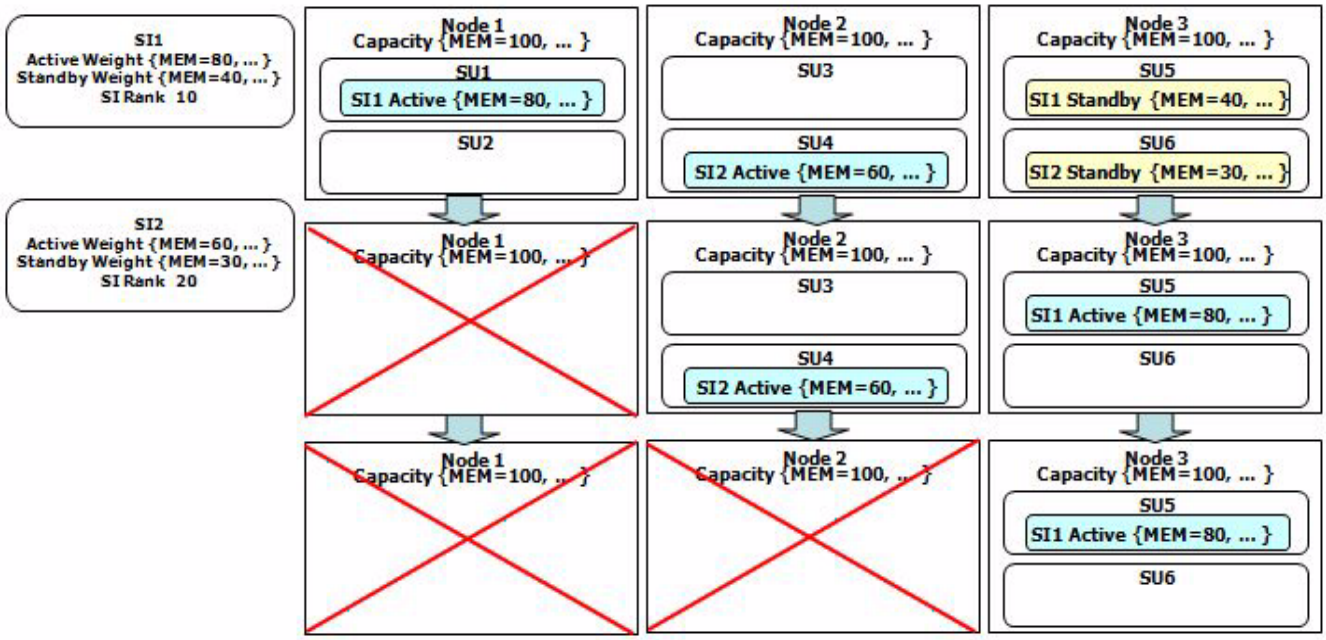
FIGURE 6 and FIGURE 7 next show two examples for node capacity limitation.

In both examples, assume that there are two service groups protecting two SIs, both service groups associated with the 2N redundancy model; SG1 contains SU1, SU3 (spare service unit), and SU5; SG2 contains SU2 (spare service unit), SU4, and SU6.

In the first example, if Node1 fails, AMF assigns SI1 as active to SU5. As with this assignment the capacity of Node3 would be exceeded, the standby assignment of SI2 to SU6 is removed.

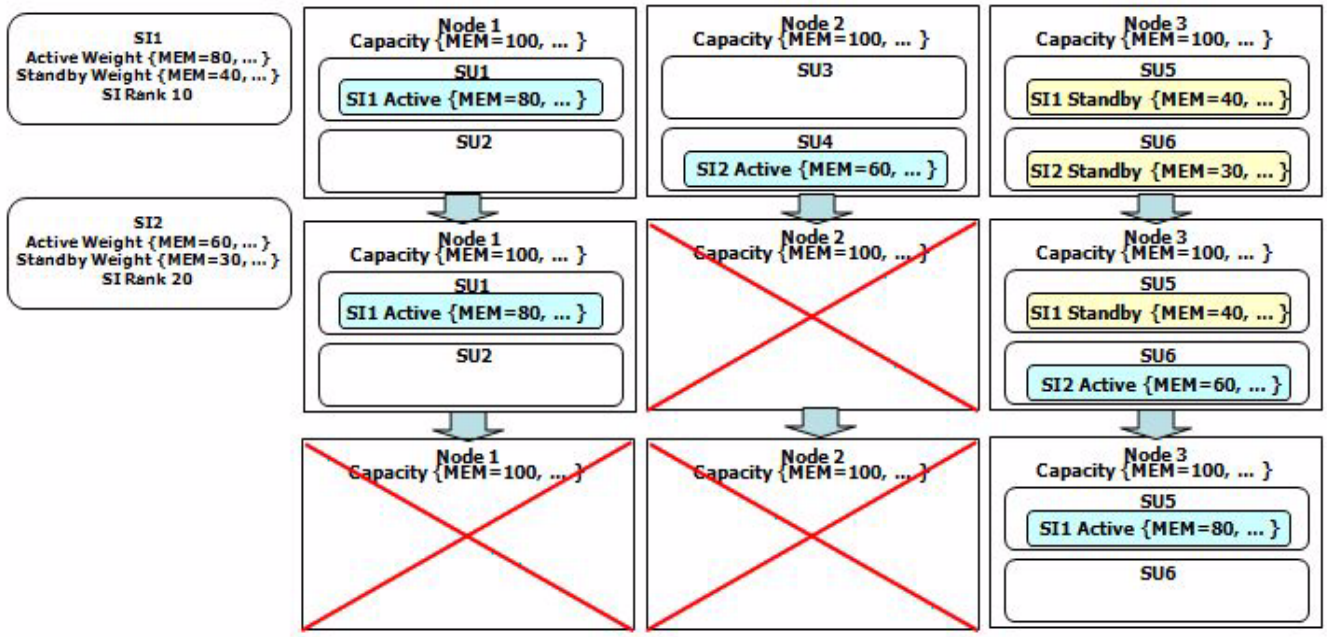
Now, assume that Node2 also fails. SI1 and SI2 cannot be both assigned as active to SU5 and SU6, respectively, because with these assignments the capacity of Node3 would be exceeded. As the rank of SI2 is lower than the rank of SI1, SI2 is not assigned as active to SU6.

FIGURE 6 Example 1 for Node Capacity Limitation



In FIGURE 7, assume that Node2 fails. Then the assignments of SI1 are = {active: SU1; standby: SU5}, and the assignments of SI2 are = {active: SU6}.
If subsequently Node1 fails too, as the capacity of Node3 does not permit both active SI1 and active SI2 assigned to it at the same time, and the rank of SI2 is lower than that of SI1, the active assignment of SI2 is removed from Node3.

FIGURE 7 Example 2 for Node Capacity Limitation



3.6.1.4 Considerations when Configuring Redundancy

This document specifies various features and options that are used by an Availability Management Framework implementation to decide which service units are instantiated and which service instances are assigned to them. However, it does not completely specify how an implementation uses the provided features and options to best provide the highest level of service in case of a shortage of resources.

In general, resources like instantiated AMF nodes having sufficient capacity to host service units with the desired service instances assigned to them must be available for the Availability Management Framework when it calculates how service units are distributed in node groups and how service instances are assigned. However, in certain cases during the lifetime of the cluster, a shortage of resources may occur.

As has been explained, it is highly recommended to perform consistency checks on the configuration, as incorrectly setting configuration options could provide the Availability Management Framework with contradictory rules for its reactions on a shortage of resources.

The following notes are intended to help the system architect to decide how to use the various configuration options. These features are, for example, the cluster-wide SI rank, the SI weight (and the limiting node capacity), the auto-adjust option for service instances, the ordered list of service units, and node groups.

Service Groups and Nodes

A service unit may be configured on a specific node. A more flexible configuration is possible by configuring a node group for a service unit or for its service group. A poor configuration may lead to nodes being idle, while other nodes cannot provide enough resources for all assignments.

Note that the assignment of a service unit to a node of the configured node group does not change dynamically. The Availability Management Framework will not reassign service unit to nodes within a node group when a node fails.

Ranks, Weights, and Capacities

Ranks, weights, and capacities are the basis for the Availability Management Framework to calculate service instance assignments within the ordered lists of service units and node groups.

SU ranks are used by the Availability Management Framework to calculate the order of assignments within a service group. SI ranks define the importance of a service instance globally in the cluster and, thus, also the importance of service groups compared to each other. SI ranks are used globally when the Availability Management Framework—due to a shortage of resources—cannot assign all service instances that are needed to support the preferred number of assignments. Cases for a shortage of resources are:

- there are not enough nodes in-service to host the service units, or
- there are not enough service units in-service that can be assigned, or
- the capacity of the nodes is not sufficient.

The configuration of SI ranks needs to take into account SI dependencies, which can be explicitly expressed with the `SaAmfSIDependency` object class in the configuration or implicitly with the proxy-proxied or container-contained component relations. That is, the rank of the SI on which other SIs depend must not be lower than the ranks of the dependent SIs, so that dropping of lower ranks SIs does not force higher-ranked, dependent SIs to also be dropped.

Auto-Adjust Option

If the auto-adjust option is specified, the Availability Management Framework reassigns service instances that are already assigned to service units to other service units when it becomes possible, so as to match the preferred configuration.

3.6.2 2N Redundancy Model

3.6.2.1 Basics

In a service group with the **2N redundancy model**, at most one service unit will have the active HA state for all service instances (usually called the active service unit), and at most one service unit will have the standby HA state for all service instances (usually called the standby service unit). Some other service units may be considered spare service units for the service group, depending on the configuration. The components in the active service unit execute the service, while the components in the standby service unit are prepared to take over the active role if the active service unit fails.

Although the goal of the 2N redundancy model is to offer redundancy in service, it is possible that a 2N redundancy service group is configured to have only one service unit. In this case, no redundancy is provided at the service units-level; however, the Availability Management Framework manages the availability of such a degenerated service group. The specification supports this single service unit 2N redundancy model, because it makes easier, from the configuration-update perspective, to add more service units later on when, for example, more nodes are configured into the cluster.

Components implementing any of the capability models described in Section 3.5 on page 107 can participate in the 2N redundancy model.

Examples of a service group with a 2N redundancy model are presented in Section 3.6.2.4 on page 125.

For the sake of simplicity, in the subsequent discussion of this redundancy model, it is assumed that the HA readiness state of all service units for all SIs is ready-for-assignment and that the resource capacity limits of the nodes are never exceeded, as the Availability Management Framework assigns and reassigns the SIs.

3.6.2.2 Configuration

- **Ordered list of service units for a service group**: this parameter is described in Section 3.6.1.1.
  Default value: no default, the order is implementation-dependent.
- **Preferred number of in-service service units at a given time**: the Availability Management Framework should make sure that this number of in-service service units is always instantiated, if possible. This preferred number is configured by setting the `saAmfSGNumPrefInserviceSUs` attribute of the `saAmfSG` object class (see Section 8.9). If the ordered list of service units of a service

group has at least two service units, then the preferred number of in-service service units should be at least two. If the preferred number of in-service service units is greater than two, the service group will contain some instantiated spare service units. These service units are called "spare" service units. The preferred number of in-service service units for the service groups containing only non-pre-instantiable components must be set to one.
Default value: two

- **Auto-adjust option**: for the general explanation of this option, refer to Section 3.6.1.1 on page 110.
Section 3.6.2.3.3 on page 123 discusses how this option is handled in this redundancy model.
Default value: no auto-adjust

3.6.2.3 SI Assignments and Failure Handling

3.6.2.3.1 Failure of the Active Service Unit

When an active service unit fails over, the associated standby service unit will be assigned active for all SIs. Then, one of the spare service units will be selected and will be assigned standby for all SIs. If the number of instantiated service units falls below the preferred number of in-service service units, another service unit from the ordered list of instantiable service units will be instantiated.

3.6.2.3.2 Failure of the Standby Service Unit

When a standby service unit fails, one of the spare service units will be assigned to take over the standby role, if possible. If the number of instantiated service units falls below the preferred number of in-service service units, another service unit from the set of instantiable service units will be instantiated.

3.6.2.3.3 Auto-Adjust Procedure

If the auto-adjust option is set in the configuration, the Availability Management Framework should make sure that the service group assignments are assigned back to the preferred configuration, meaning that the highest-ranked in-service service unit be active and the second highest-ranked in-service service unit be standby. It is obvious that the auto-adjust procedure may involve relocation of SIs. Though it is left to the implementation how to perform an auto-adjust, it should be done with minimum impact on the availability of the corresponding service.

3.6.2.3.4 Cluster Startup

Because the cluster startup is a rare event, its latency may not be as critical as the latency of other failure recovery events such as a service unit fail-over. Moreover, it is very important to start a cluster in an orderly fashion, so that the initial runtime status of the entities under the Availability Management Framework's control is as close as possible to the preferred configuration. Saying so, during the startup of the cluster, the Availability Management Framework should wait for at most a predefined period of time to make sure that all required service units are instantiated before assigning SIs to service units. This period of time is specified in the `saAmfClusterStartupTimeout` configuration attribute of the `SaAmfCluster` object class, shown in Section 8.7. It is left to the implementation how to handle cluster startup; however, the implementation should make sure that the initial assignments are as close as possible to the preferred assignments.

3.6.2.3.5 Role of the List of Ordered Service Units in Assignments and Instantiations

The ordered list of service units determines the preferred configuration: the highest-ranked in-service service unit should be assigned active and the second highest-ranked in-service service unit assigned standby. It is used for the following purposes:

⇒ To choose among all instantiable service units which one must be instantiated. This choice is made in situations like:

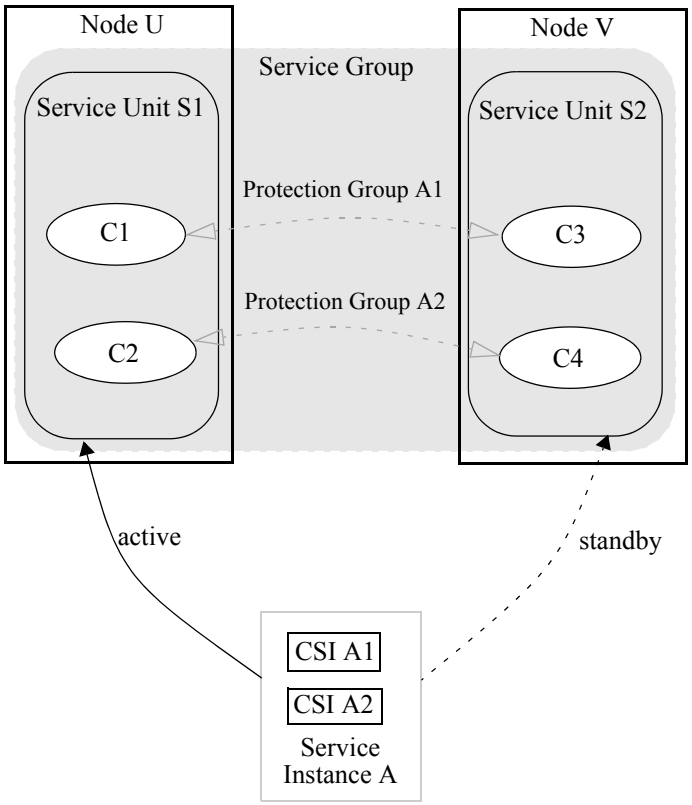
- when the number of instantiated service units drops below the preferred number, and new service units must be instantiated, and
- when an auto-adjust procedure is performed or the `SA_AMF_ADMIN_SG_ADJUST` administrative operation is executed, and the current situation does not match the preferred configuration.

⇒ To select which of the instantiated service units will have active and standby assignments. This choice is made each time a service instance must be assigned to an in-service service unit.

3.6.2.4 Examples

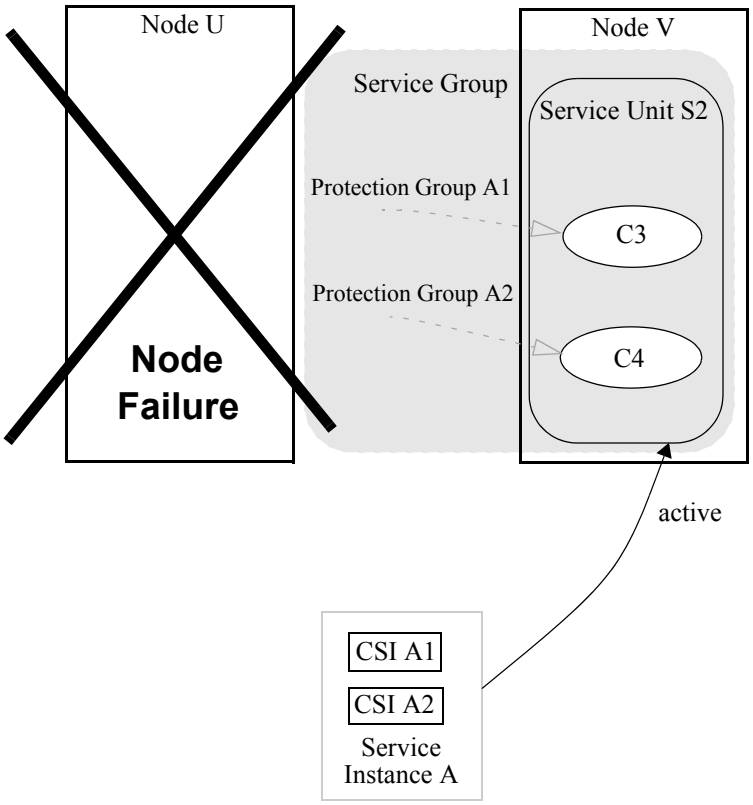
In the following example, it is assumed that the number of preferred in-service service units is set to 2.

FIGURE 8 Example of the 2N Redundancy Model: Two Service Units on Different Nodes



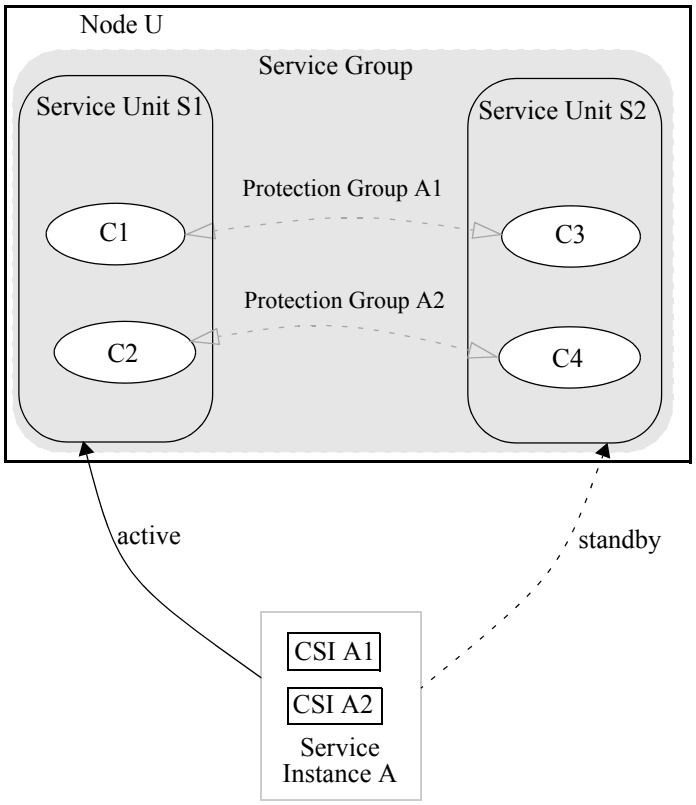
After a fault that disables node U, service unit S2 on node V will be assigned to be active for service instance A, as shown in FIGURE 9.

FIGURE 9 Example of the 2N RM. Two SUs on Different Nodes, Fault Has Occurred



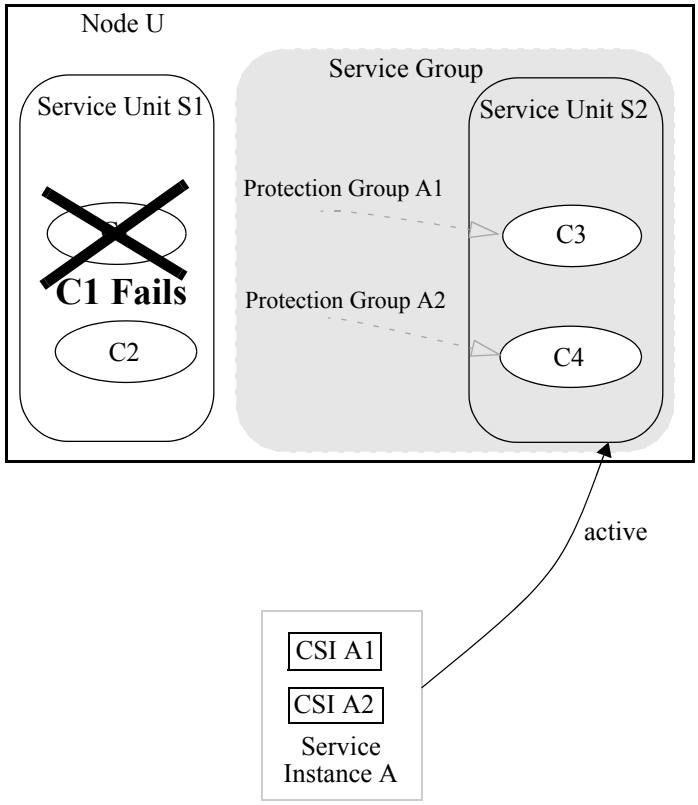
The two service units may even reside on the same node, as shown in FIGURE 10, which allows one to implement software redundancy with two instances of the application running on the same node.

FIGURE 10 Example of the 2N Redundancy Model: Two Service Units on the Same Node



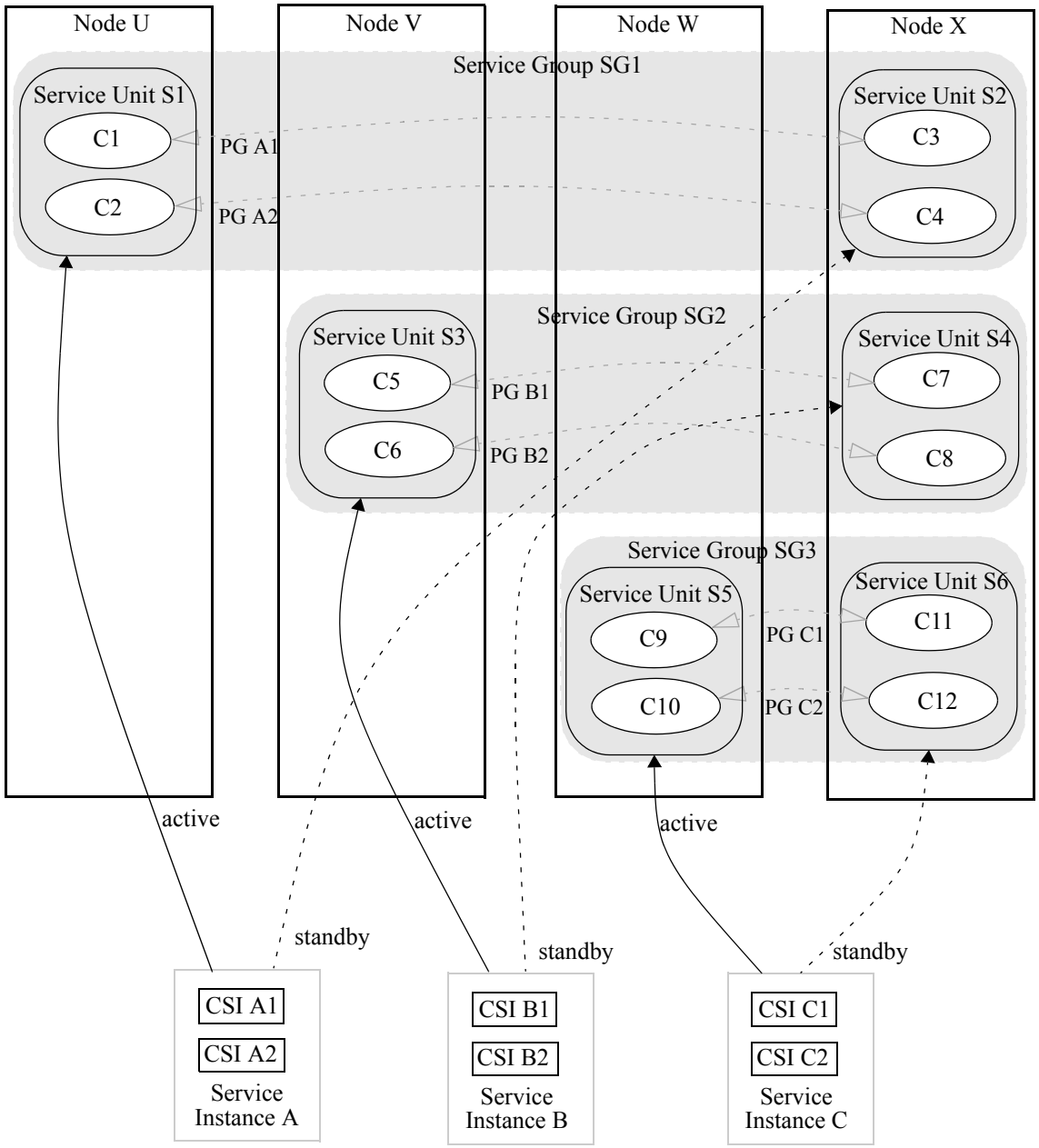
As shown in FIGURE 11, after a fault that disables component C1 within service unit S1, service unit S2 is assigned to be active for service instance A. Note that a fault that affects any component within a service unit and that cannot be recovered by restarting the affected component causes the entire service unit and all components within the service unit to be withdrawn from service. In this example, even though component C2 is still fully operational, it must fail-over to component C4.

FIGURE 11 Example of the 2N RM: Two SUs on the Same Node, Fault Has Occurred



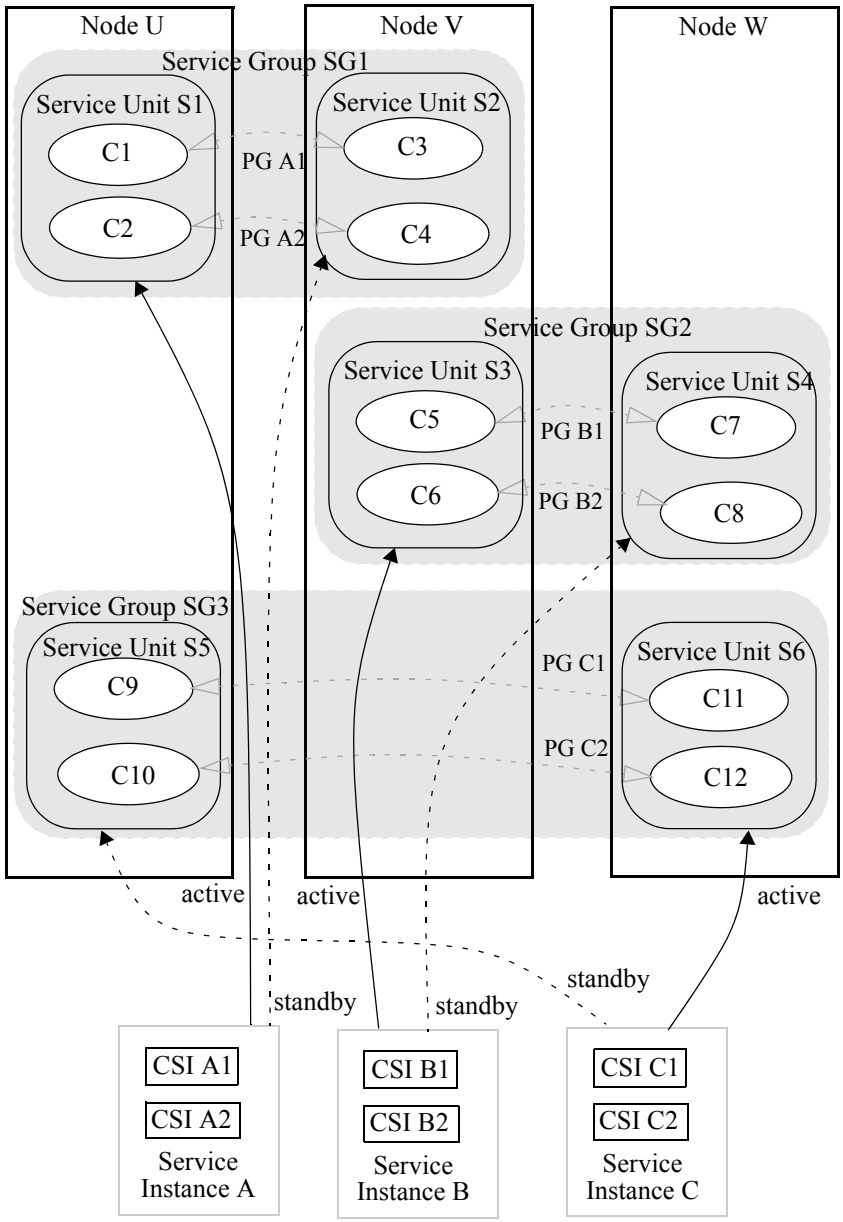
As shown in the FIGURE 12, the 2N service group redundancy model can support N+1 strategies at the node level. Node X supports standby service units for several service groups. If one of the other nodes fails, the corresponding service unit on node X will be reassigned to be active for the service instance supported by the failed node. Note that node X must support multiple service units, and might require additional resources like memory.

FIGURE 12 Example of the 2N RM: One Node Provides Standby SUs for Several Service Groups



As FIGURE 13 illustrates, the 2N redundancy model can also support strategies in which all nodes host some service units that are active for their service instances and other service units that are standby for their service instances.

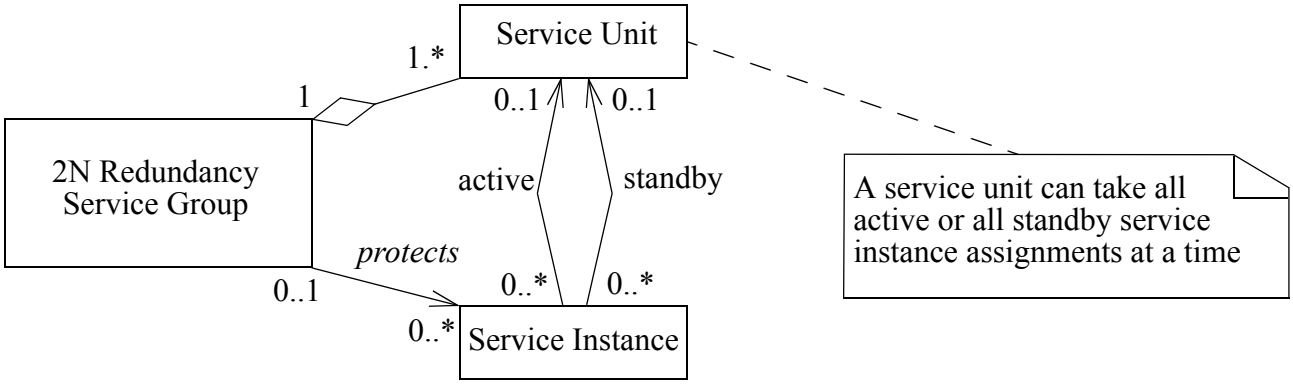
FIGURE 13 Example of the 2N RM: Each Node Has an Active and a Standby Service Unit



3.6.2.5 UML Diagram of the 2N Redundancy Model

The 2N redundancy model is represented by the UML diagram shown in FIGURE 14.

FIGURE 14 UML Diagram for the 2N Redundancy Model



3.6.3 N+M Redundancy Model

3.6.3.1 Basics

The **N+M redundancy model** extends the 2N redundancy model by allowing more than two service units to have active or standby service instance assignments. N service units have the active assignments and M service units have the standby assignments.

For the sake of simplicity, in the subsequent discussion of this redundancy model, it is assumed that the HA readiness state of all service units for all SIs is ready-for-assignment and that the resource capacity limits of the nodes are never exceeded, as the Availability Management Framework assigns and reassigns the SIs.

This redundancy model has the following characteristics:

- A service unit can be
  (i) active for all SIs assigned to it or
  (ii) standby for all SIs assigned to it.
  In other words, a service unit cannot be active for some SIs and standby for some other SIs at the same time.
- At any given time, several in-service service units can be instantiated for a service group: some service units are active for some SIs, some service units are standby for some SIs, and possibly some other service units are considered spare service units for the service group. For simplicity of the discussion, the service units having the active HA state for all SIs assigned to them are denoted as "active service units", and the service units having the standby HA state for all SIs assigned to them are denoted as "standby service units".
- The number of active service units, the number of standby service units, and the number of spare service units of a service group are dynamic and can change during the life-span of the service group; however, the preferred number of these service units can be configured, as discussed in Section 3.6.3.3 on page 135.
- For each SI and at any given time, there will be at most one active service unit and at most one standby service unit.
- At any given time, the Availability Management Framework should make sure that the per-SI redundancy level (one service unit assigned the active HA state and a service unit on another node assigned the standby HA state for each SI) is guaranteed, while requirements on the load constraints in each service unit and the number of available spare service units (see Section 3.6.3.3) are fulfilled.

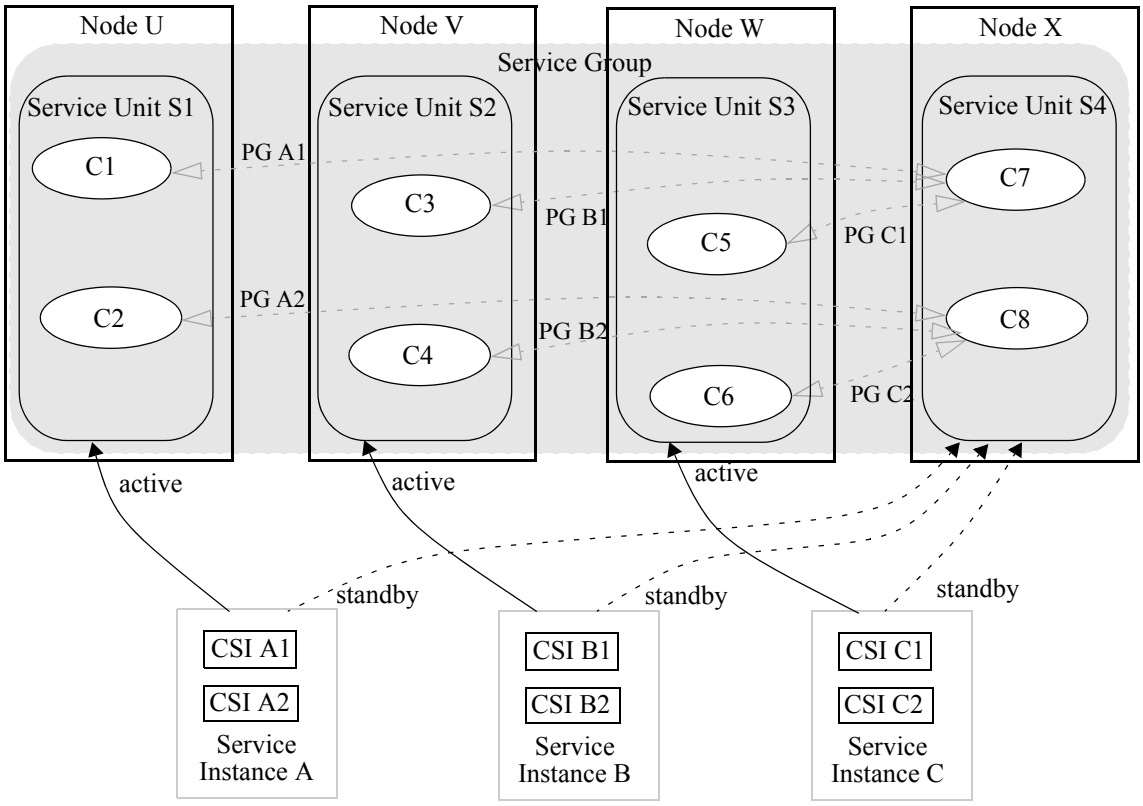
- As mentioned before, the objective should be to maintain the redundancy level for all SIs (one service unit assigned the active HA state and another service unit assigned the standby HA state for each SI); however, this may not be feasible in some cases due to a shortage of available service units for the service group. For example, if the number of in-service service units is not large enough to support full redundancy levels for all SIs, then some of the SIs could be supported in a degraded mode (for instance, no service unit assigned standby for this SI). The order of importance of SIs can be configured, as discussed in Section 3.6.3.3.

Components implementing any of the capability models described in Section 3.5 on page 107, except the 1_active _or_1_standby capability model, can participate in the N+M redundancy model.

3.6.3.2 Examples

A common use of the N+M redundancy model is the N+1 redundancy model, in which a single service unit is assigned standby for N active service units, as shown in FIGURE 15. The following diagram depicts a typical N+1 configuration. Note that each of the components C7 and C8 of the standby service unit supports three component service instances. Node X might require additional resources like memory to accommodate additional component service instances.

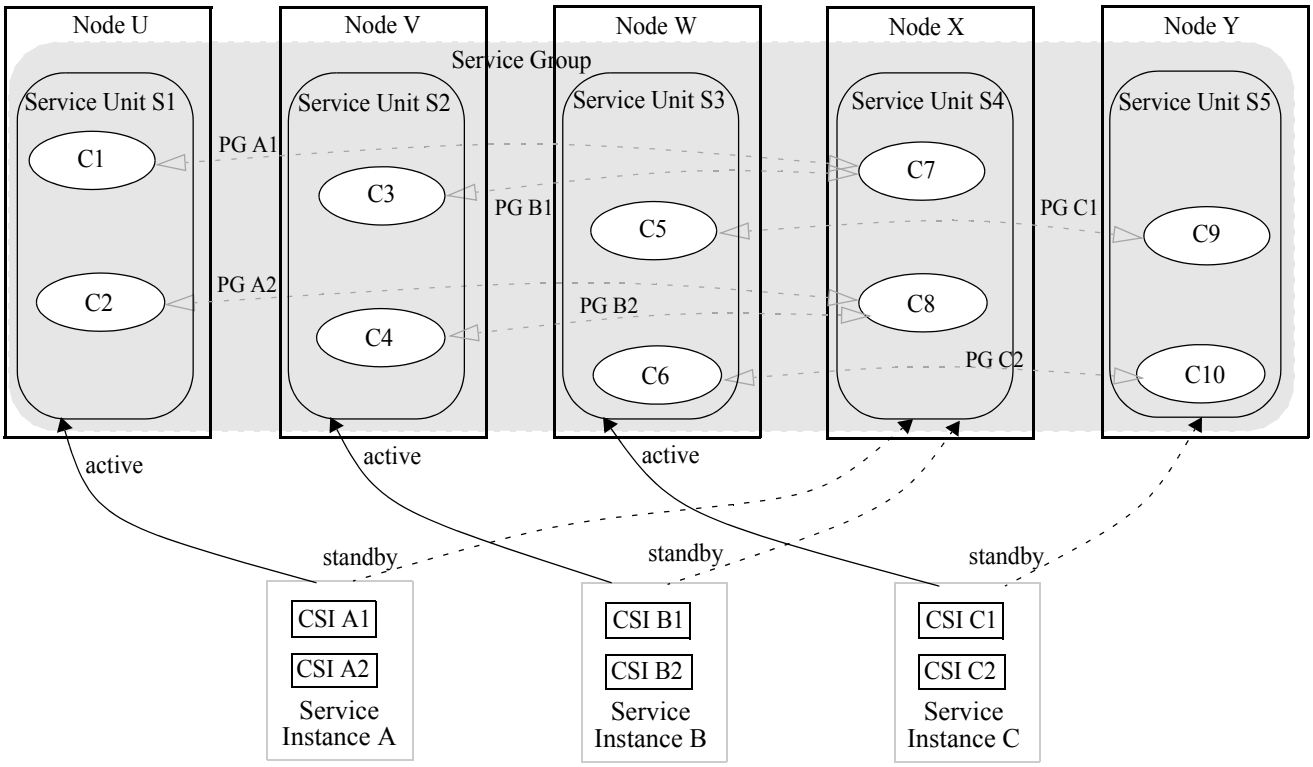
FIGURE 15 Example of the N+1 Redundancy Model



To illustrate a fail-over in the N+M model, assume that the service unit S2 fails. As a consequence, service unit S4 should be assigned the active HA state for SI B. As S4 must not be assigned active for some SIs and standby for other SIs at the same time in accordance with the redundancy model, the standby HA state for service instances A and C will be removed from S4. Note that this scenario also applies if the involved component capability models are x_active_and_y_standby.

In a more general N+M case, the M standby service units can be freely associated with the N active service units. FIGURE 16 shows an example of the N+M redundancy model with N=3 and M=2.

FIGURE 16 Example of the N+M Redundancy Model, Where N = 3 and M = 2



3.6.3.3 Configuration

- **Ordered list of service units for a service group**: this parameter is described in Section 3.6.1.1.
  Default value: no default, the order is implementation-dependent.
- **Ordered list of SIs**: for the general meaning of this parameter, refer to its definition in Section 3.6.1.1. The Availability Management Framework uses this ranking to select some SIs to support either in non-redundant mode (that is, for each of these SIs, there is a service unit having the active HA state, but no service unit having the standby HA state) or to drop them completely if the Availability Management Framework encounters a shortage of service units for the full support of all SIs; however, it is important to note that the Availability Management Framework should consider not only the ordering of the SIs but also their dependencies when it chooses some SIs to support partially or to drop them.
  Default value: no default, the order is implementation-dependent.

- **Preferred number of in-service service units**: the Availability Management Framework should make sure that this number of in-service service units is always instantiated, if possible. This preferred number is configured by setting the `saAmfSGNumPrefInserviceSUs` attribute of the `saAmfSG` object class (see Section 8.9). If the service units list for a service group includes at least two service units, then the preferred number of instantiated service units should be at least two.
  Default value: the number of configured service units for the service group.
- **Preferred number of active service units**: this parameter indicates the preferred number of active service units at any time. This preferred number is configured by setting the `saAmfSGNumPrefActiveSUs` attribute of the `saAmfSG` object class (see Section 8.9). The Availability Management Framework should guarantee that this number of active service units exist for the service group provided that the number of in-service service units is large enough.
  Default value: no default value is specified. It is mandatory to set this number for each service group.
- **Preferred number of standby service units**: this indicates the preferred number of standby service units at any time. This preferred number is configured by setting the `saAmfSGNumPrefStandbySUs` attribute of the `saAmfSG` object class (see Section 8.9). The Availability Management Framework should guarantee that this number of standby service units exist for the service group provided that the number of in-service service units and the number of service units associated with the service group are large enough.
  Default value: no default value is specified. It is mandatory to set this number for each service group.
- **Maximum number of active SIs per service unit**: this indicates the maximum number of SIs that can be assigned to a service unit, so that the service unit has the active HA state for all these SIs. It is assumed that the load imposed by each SI is the same. If this assumption is not true for some service instances, the service deployer has to approximate. This maximum number is configured by setting the `saAmfSGMaxActiveSIsperSU` attribute of the `saAmfSG` object class (see Section 8.9).
  Default value: no limit, a value of 0 is used to specify this.
- **Maximum number of standby SIs per service unit**: this indicates the maximum number of SIs that can be assigned to a service unit, so that the service unit has the standby HA state for all these SIs. It is assumed that the load imposed by each SI is the same. This maximum number is configured by setting the `saAmfSGMaxStandbySIsperSU` attribute of the `saAmfSG` object class (see Section 8.9).
  Default value: no limit, a value of 0 is used to specify this.

- **Auto-adjust option**: for the general explanation of this option, refer to Section 3.6.1.1 on page 110.
  Section 3.6.3.6 on page 146 shows an example for handling the auto-adjust option in this redundancy model.
  Default value: no auto-adjust

3.6.3.4 SI Assignments

In this section, the general direction in assigning SIs to in-service service units is discussed. Then, the assignment procedure will be illustrated using example configurations.

If available service units for the service group allow it, the Availability Management Framework will instantiate the preferred number of in-service service units for the service group. Then, as many service units as the preferred number of active service units will be assigned the active HA state for SIs, and as many service units as the preferred number of standby service units will be assigned the standby HA state for SIs, according to the configuration. Additionally, some of the service units will be dedicated as spare.

It is assumed that the service group configuration has passed a series of validations, so that when as many service units as the preferred number of active service units are assigned the active HA state, and as many service units as the preferred number of standby service units are assigned the standby HA state, one service unit will be assigned the active HA state, and another service unit will be assigned the standby HA state for each SI of the service group, without violating the load limits expressed in Section 3.6.3.3.

In case of a shortage of in-service service units, the Availability Management Framework should use the ordered list of SIs in choosing which SIs have to be dropped or supported in non-redundant mode (that is, for each of these SIs, there is a service unit having the active HA state, but no service unit having the standby HA state).

In the remainder of this section, the SI assignment procedure is described. The following example of a service group configuration will be used throughout this illustration:

- Ordered list of service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- Ordered list of SIs = {SI1, SI2, SI3, SI4, SI5, SI6}
- Preferred number of in-service service units = 7
- Preferred number of active service units = 3
- Preferred number of standby service units = 3

- Maximum number of active SIs per service unit = 3
- Maximum number of standby SIs per service unit = 4

Assignment I: Full Assignment with Spare Service Units

As an initial example, it is assumed that all service units of the preceding configuration can be brought in-service. Then, the following can be a running configuration for the service group.

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {SU8}
- active service units = {SU1, SU2, SU3}
- standby service units = {SU4, SU5, SU6}
- spare service units = {SU7}

Then, the assignments look like:

- SIs assigned to SU1 as active = {SI1, SI2}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as standby = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby ={SI5, SI6}

The following points should be mentioned regarding the assignments:

(1) The selection of instantiated, active, and standby service units is based on the ordered list of service units.

(2) The assignments of SIs to service units are based on the ordered list of SIs.

(3) Service units are not fully used to their capacities. Each active service unit could handle one more SI. Similarly, each standby service unit can handle two more SIs. This extra slack will be used in case of a shortage of service units due to the unavailability of some nodes.

Note: This specification does not define the actual algorithm for SI assignments; instead, it provides rules and examples to guide implementers. The examples provided are only illustrative and represents one possible assignment scenario (by a particular implementation) based on the configuration specified in Section 3.6.3.4. Implementers should design their own assignment algorithms by following the given rules.

The difficulty comes when the number of in-service service units is not enough to satisfy the configuration requirements. The first goal is to try to keep all SIs in the redundant mode (that is, for each of these SIs, one service unit has the active HA state, and another service unit has the standby HA state), even at the expense of imposing maximum load on each service unit. If this goal is not attainable, the next goal is to keep as many SIs as possible in a redundant mode while all SIs are assigned active in one of the service units. This procedure may lead to a reduction in the number of standby service units. Finally, if this objective is also not attainable, the only choice is to drop some of the SIs completely. This means reducing further the number of active service units.

The following subsections sketch the procedure for assigning service units and SIs in situations of shortage of in-service service units.

3.6.3.4.1 Reduction Procedure

The following procedure is for assigning SIs to in-service service units and for supporting the N+M service group if not enough service units are available.

If the number of in-service service units is not large enough to support the preferred number of active, standby, and spare service units, as defined in the configuration, the following procedure is used to maintain an acceptable level of support for the service group.

Step 1: Reduction of the Number of Spare Service Units

If the number of instantiated service units does not allow enough spare service units, the service group should be maintained with less spare service units than the required number. The number of the spare service units is reduced until:

(1.a) The Availability Management Framework succeeds in allocating the preferred number of active and standby service units. In this case, the assignment procedure is completed.

OR

(1.b) After dropping all spare service units, the Availability Management Framework does not succeed in allocating the preferred number of active and standby service units. In this case, the assignment procedure continues to the next step ((2.a) or (2.b)).

The following example illustrates case (1.a).

Assignment II: Full Assignment with Spare Reduction

Assume that the state of the cluster is as follows:

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6}
- instantiable service units = {}
- active service units = {SU1, SU2, SU3}
- standby service units = {SU4, SU5, SU6}
- spare service units = {}

Based on the preceding configuration, SI assignments fulfilling the condition that every SI is in redundant mode can be:

- SIs assigned to SU1 as active = {SI1, SI2}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as standby = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6}

Step 2: Reduction of the Number of Standby Service Units

If the preferred number of active and standby service units cannot be supported due to a shortage of in-service service units, the Availability Management Framework is forced to use fewer standby service units than the preferred number expressed in the configuration. As the number of standby service units gets smaller, the number of SIs assigned to each standby service units increases. The Availability Management Framework needs to guarantee that the load does not exceed the service units capacity expressed in the configuration.

The number of standby service units is reduced until:

(2.a) The preferred number of active service units is reached, and, for each SI, a service unit has been assigned the standby HA state without violating the capacity levels of the service units. In this case, the assignment procedure is completed.

OR

(2.b) All standby service units have been loaded to their maximum capacity, but some SIs are still without standby assignments. In this case, the assignment procedure continues to the next step ((3.a) or (3.b)).

The following example illustrates case (2.a).

Assignment III: Full Assignment With Reduction of Standby Service Units

Assume that the state of the cluster is such that the only service units that can be brought in-service are SU1, SU2, SU3, SU4, and SU5.

These instantiated service units take the following responsibilities:

- in-service service units = {SU1, SU2, SU3, SU4, SU5}
- active service units = {SU1, SU2, SU3}
- standby service units = {SU4, SU5}
- spare service units = {}

Based on the preceding configuration, SI assignments fulfilling the condition that every SI is in redundant mode can be:

- SIs assigned to SU1 as active = {SI1, SI2}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as standby = {SI1, SI2, SI3}
- SIs assigned to SU5 as standby = {SI4, SI5, SI6}

Step 3: Reduction of the Number of Active Service Units

If even after loading standby service units to their full capacity, the number of in-service service units is still not enough to maintain the preferred number of active service units, the Availability Management Framework tries to reduce the number of active service units by loading active service units to their full capacity. In this step, the number of active service units should be reduced until:

(3.a) For each SI, there is an active assignment without violating the capacity levels of active service units. In this case, the assignment procedure is completed.

OR

(3.b) All active service units have been loaded to their maximum capacity, but some SIs are still without active or standby assignments. In this case, the assignment procedure should continue to the next step ((4.a) or (4.b)).

The following example illustrates case (3.a).

Assignment IV: Full Assignment with Reduction of Active Service Units

Assume that the state of the cluster is such that the only service units that can be brought in-service are SU1, SU2, SU3, and SU4.

These instantiated service units take the following responsibilities:

- in-service service units = {SU1, SU2, SU3, SU4}
- active service units = {SU1, SU2}
- standby service units = {SU3, SU4}
- spare service units = {}

Based on the preceding configuration, the SI assignments can be:

- SIs assigned to SU1 as active = {SI1, SI2, SI3}
- SIs assigned to SU2 as active = {SI4, SI5, SI6}
- SIs assigned to SU3 as standby = {SI1, SI2, SI3}
- SIs assigned to SU4 as standby = {SI4, SI5, SI6}

Note that in the preceding assignments, all SIs are still supported in redundant mode.

Step 4: Reduction of the Standby Assignments for some SIs

At this step of the assignment procedure, the number of instantiated service units is not enough to guarantee redundant assignments for all SIs; therefore, the Availability Management Framework is forced to drop the standby assignment of some SIs. The Availability Management Framework will use the ordered SI list to decide for which SIs standby assignments should be dropped. The standby assignments for some SIs will be dropped until:

(4.a) For each SI, there is a service unit with the active HA state for this SI. In this case, the assignment procedure is completed.

OR

(4.b) The number of the in-service service units is so small that the Availability Management Framework cannot assign the active HA state to these service unit for all SIs. In this case, the reduction procedure continues to the next step (5).

The following example illustrates case (4.a).

Assignment V: Partial Assignment with Reduction of Standby Assignments

Assume that the state of the cluster is such that only the service units SU1, SU2, and SU3 can be brought in-service.

The instantiated service units take the following responsibilities:

- in-service service units = {SU1, SU2, SU3}
- active service units = {SU1, SU2}
- standby service units = {SU3}
- spare service units = {}

Based on the preceding configuration, the SI assignments can be:

- SIs assigned to SU1 as active = {SI1, SI2, SI3}
- SIs assigned to SU2 as active = {SI4, SI5, SI6}
- SIs assigned to SU3 as standby = {SI1, SI2, SI3, SI4}

In this assignment, SI5 and SI6 are supported only in non-redundant mode (that is, for each of these SIs, there is a service unit having the active HA state, but no service unit having the standby HA state).

Step 5: Reduction of the Active Assignments for some SIs

At this stage of the reduction procedure, the number of instantiated service units is so small that the Availability Management Framework cannot guarantee that service units have been assigned active for all SIs. Therefore, some of the SIs should be dropped. As stated earlier, the ordered list of SIs should be used to decide which SIs should be dropped. This last step continues until a subset of the SIs are supported in non-redundant mode (that is, for each of these SIs, there is a service unit having the active HA state, but no service unit having the standby HA state).

The following example illustrate the last step of the reduction procedure.

Assignment VI: Partial Assignment with SIs Drop-Outs

Assume that the state of the cluster is such that SU1 is the only service unit that can be brought in-service.

The instantiated service units take the following responsibilities:

- in-service service units = {SU1}
- active service units = {SU1}
- standby service units = {}
- spare service units = {}

Based on the preceding configuration, the SI assignments can be:

- SIs assigned to SU1 as active = {SI1, SI2, SI3}

Note that in the preceding example, SI4, SI5, and SI6 are completely dropped.

3.6.3.5 Examples for Service Unit Fail-Over

The Availability Management Framework should handle failures in a way that the availability of all SIs supported by service groups are guaranteed, if possible. The following examples should be considered as illustrations of high-level requirements on the Availability Management Framework failure handling and should not be seen as the only way of failure handling.

3.6.3.5.1 Handling of a Node Failure when Spare Service Units Exist

Assume the following cluster configuration before the node hosting SU1 failed:

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- active service units = {SU1, SU2, SU3}
- standby service units = {SU4, SU5, SU6}
- spare service units = {SU7}
- SIs assigned to SU1 as active = {SI1, SI2}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as standby = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6}

When the node hosting SU1 fails, SI1 and SI2 lose their active assignments; therefore, the Availability Management Framework must react in attempting to restore the active assignments for SI1 and SI2. This attempt is the immediate reaction of the Availability Management Framework to the failure. Additionally, the Availability Management Framework should use the spare service unit to restore the standby assignment for SI1 and SI2 as well. After the recovery, the assignment should look like as follows:

- in-service service units = {SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- active service units = {SU2, SU3, SU4}
- standby service units = {SU5, SU6, SU7}
- spare service units = {}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as active = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6}
- SIs assigned to SU7 as standby = {SI1, SI2}

3.6.3.5.2 Handling of a Node Failure when no Spare Service Units Exist

The following example illustrates how the Availability Management Framework uses the available capacity of service units to retain the redundant mode of SIs when a node hosting some service units fails.

Assume the following cluster configuration before the failure of the node hosting SU2:

- in-service service units = {SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- active service units = {SU2, SU3, SU4}
- standby service units = {SU5, SU6, SU7}
- spare service units = {}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as active = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}

- SIs assigned to SU6 as standby = {SI5, SI6}
- SIs assigned to SU7 as standby = {SI1, SI2}

When the node hosting SU2 fails, SI3 and SI4 lose their active assignments; therefore, the immediate action for the Availability Management Framework is to restore the active assignments of SI3 and SI4. Additionally, the standby assignments of these SIs should also be restored. A couple of different ways exist for restoring the standby assignments for SI3 and SI4. It depends on the Availability Management Framework implementation how to restore the standby assignments without violating the configuration parameters (such as the number of active/standby SIs assigned to a service unit).

One way of restoring the standby assignments for SI3 and SI4 is the following one.

- in-service service units = {SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- active service units = {SU3, SU4, SU5}
- standby service units = {SU6, SU7}
- spare service units = {}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as active = {SI1, SI2}
- SIs assigned to SU5 as active = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6, SI4}
- SIs assigned to SU7 as standby = {SI1, SI2, SI3}

3.6.3.6 Example of Auto-Adjust

The auto-adjust option indicates that the current (running) configuration of the service group needs to return to the preferred configuration, so that the service units with the highest ranks are active and the highest-ranked SIs are assigned in redundant mode (that is, there is a service unit having the active HA state for each of these SIs and another service unit having the standby HA state for each of these SIs). It is up to the Availability Management Framework implementation to decide when and how the auto-adjust will be initiated. The following example is given for illustration purposes. Assume that the following is the configuration of the service group.

- in-service service units = {SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- active service units = {SU2, SU3, SU4}
- standby service units = {SU5, SU6, SU7}
- spare service units = {}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as active = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6}
- SIs assigned to SU7 as standby = {SI1, SI2}

Now, assume that the node hosting SU1 joins the cluster. As a result, SU1 becomes instantiable. Because SU1 has the highest rank in the ordered list of service units, the preceding configuration is no longer a preferred one. The auto-adjust is initiated in a implementation-dependent way. After the completion of the auto-adjust procedure (assuming that SU1 could be brought in-service) the service group configuration should look like as follows:

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- active service units = {SU1, SU2, SU3}
- standby service units = {SU4, SU5, SU6}
- spare service units = {SU7}

The assignments look like:

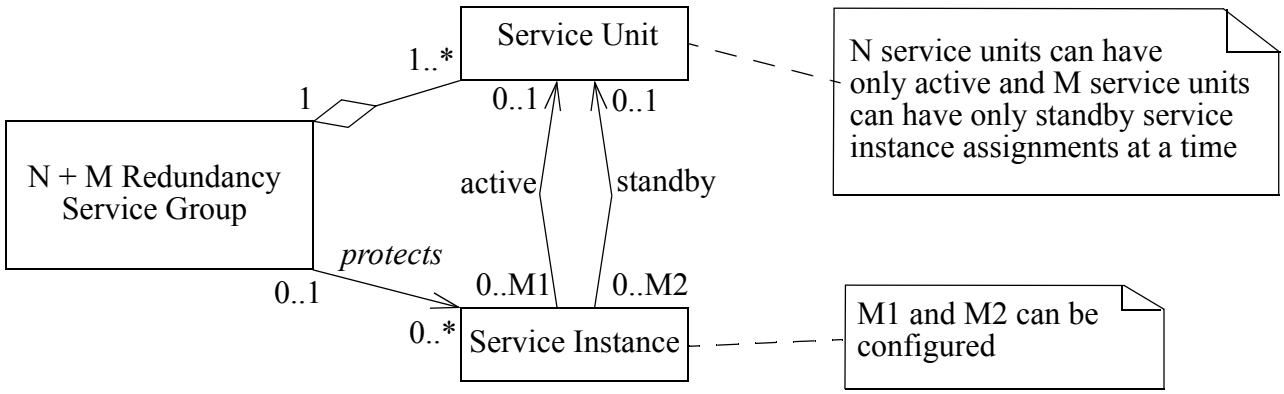
- SIs assigned to SU1 as active = {SI1, SI2}
- SIs assigned to SU2 as active = {SI3, SI4}
- SIs assigned to SU3 as active = {SI5, SI6}
- SIs assigned to SU4 as standby = {SI1, SI2}
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6}

Note that the Availability Management Framework may undergo a series of SI relocations to transition from the configuration before the auto-adjust to the preceding configuration.

3.6.3.7 UML Diagram of the N+M Redundancy Model

The N+M redundancy model is represented by the UML diagram shown in FIGURE 17.

FIGURE 17 UML Diagram of the N+M Redundancy Model



3.6.4 N-Way Redundancy Model

3.6.4.1 Basics

The **N-way redundancy model** extends the N+M redundancy model allowing a service unit to have simultaneously active and standby assignments for different service instances. It has the advantage that all service units can be used to provide active service while still providing standby protection.

For the sake of simplicity, in the subsequent discussion of this redundancy model, it is assumed that the HA readiness state of all service units for all SIs is ready-for-assignment and that the resource capacity limits of the nodes are never exceeded, as the Availability Management Framework assigns and reassigns the SIs.

This redundancy model has the following characteristics:

- In a service group with the N-way redundancy model, a service unit can simultaneously be assigned
  (i) the active HA state for some SIs and
  (ii) the standby HA state for some other SIs.
- At most one service unit may have the active HA state for an SI, and none, one, or multiple service units may have the standby HA state for the same SI.
- The preferred number of standby assignments for an SI is an SI-level configuration parameter. The preferred number of standby assignments may differ for each SI.
- At any given time, several service units can be in-service for a service group: some have SI assignments and possibly some others are considered spare service units for the service group. The number of assigned service units and the number of spare service units are dynamic and can change during the life-span of the service group; however, the preferred number of these service units can be configured, as will be discussed in Section 3.6.4.3.
- At any given time, and if resources allow, the Availability Management Framework should ensure that the redundancy level is guaranteed for each SI (one service unit assigned active and as many service units as the preferred number of standby assignments assigned standby) while the load constraints in each service unit and the number of spare service units are fulfilled.
- Each SI has an ordered list of service units to which the SI can be assigned. As any service unit in a service group is capable of providing any SI defined in the configuration for the service group (see Section 3.1.6), the ordered list of service units per SI includes all the service units configured for the service group. In other words, a partial list of service units is an invalid configuration. If the number

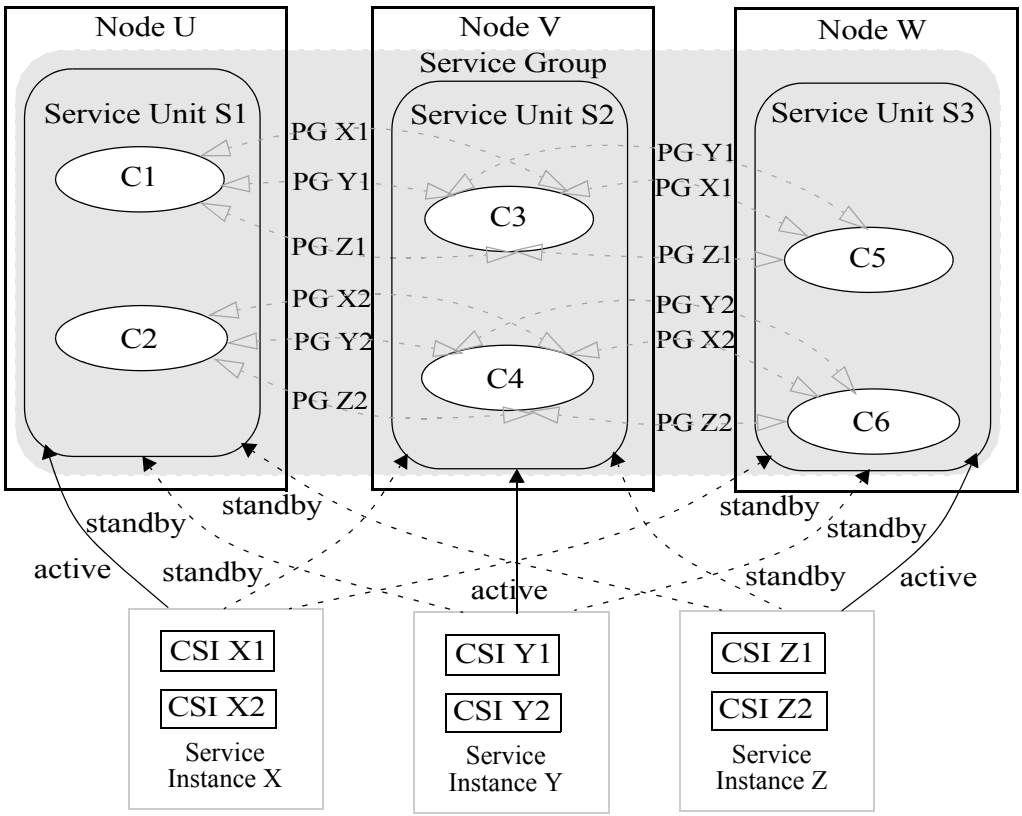
of in-service service units allows it, the Availability Management Framework should make sure that the highest-ranked in-service service units be assigned active for each service instance, and, according to the preferred number of standby assignments, the higher-ranked amongst in-service service units be assigned standby for that service instance.

Only components implementing the x_active_and_y_standby component capability model can participate in the N-way redundancy model.

3.6.4.2 Example

FIGURE 18 shows an example of the N-way redundancy model. Note that each component has the active HA state for one component service instance and the standby HA state for the other two component service instances.

FIGURE 18 Example of the N-Way Redundancy Model



3.6.4.3 Configuration

- **Ordered list of service units for a service group**: this parameter is described in Section 3.6.1.1.
  Default value: no default, the order is implementation-dependent.
- **Ordered list of SIs**: for the general meaning of this parameter, refer to its definition in Section 3.6.1.1. The Availability Management Framework uses this ranking to choose SIs to support either in non-redundant mode (that is, there is a service unit having the active HA state for each of these SIs, but no service unit having the standby HA state for each of these SIs) or to drop them completely if the set of instantiated service units does not allow full support of all SIs.
  Default value: no default, the order is implementation dependent.
- **Ranked service unit list per SI**: each SI has an ordered list of service units to which the SI can be assigned. The rank of a service unit for an SI is configured by setting the `saAmfRank` attribute of a service unit identified by `safRankedSu` in the `SaAmfSIRankedSU` association class (see Section 8.11). The rank is represented by a positive integer. The lower the integer value, the higher the rank. The Availability Management Framework should make sure that the highest-ranked available service unit be assigned active for the SI, and the remaining available high-ranked service units be assigned standby for the SI, if possible; that is, the second highest-ranked service unit is assigned the first ranked standby, the third highest-ranked service unit is assigned the second ranked standby, and so on.
  Default value: the ordered service units list defined for the service group.
- **Preferred number of standby assignments per SI**: this parameter indicates the preferred number of service units that are assigned the standby HA state for this SI. This preferred number is configured by setting the `saAmfSIPrefStandbyAssignments` attribute of the `saAmfSI` object class (see Section 8.11).
  Default value: 1
- **Preferred number of in-service service units**: the Availability Management Framework should make sure that this number of in-service service units is always instantiated, if possible. This preferred number is configured by setting the `saAmfSGNumPrefInserviceSUs` attribute of the `saAmfSG` object class (see Section 8.9). If the service units list for a service group includes at least two service units, the preferred number of in-service service units should be at least two.
  Default value: the number of the service units configured for the service group.

- **Preferred number of assigned service units**: this parameter indicates the preferred number of assigned service units at any time. This preferred number is configured by setting the `saAmfSGNumPrefAssignedSUs` attribute of the `saAmfSG` object class (see Section 8.9). As to be discussed in Section 3.6.4.4 on page 152, the Availability Management Framework should guarantee that this number of assigned service units exist for the service group provided that the number of instantiated service units is large enough.
  Default value: the preferred number of in-service service units.
- **Maximum number of active SIs per service unit**: this parameter indicates the maximum number of SIs that can be concurrently assigned to a service unit, so that the service unit has the active HA state for all these SIs. It is assumed that the load imposed by each SI is the same. This maximum number is configured by setting the `saAmfSGMaxActiveSIsperSU` attribute of the `saAmfSG` object class (see Section 8.9).
  Default value: no limit, a value of 0 is used to specify this.
- **Maximum number of standby SIs per service unit**: this parameter indicates the maximum number of standby SIs that can be concurrently assigned to a service unit, so that the service unit has the standby HA state for all these SIs. It is assumed that the load imposed by each SI is the same. This maximum number is configured by setting the `saAmfSGMaxStandbySIsperSU` attribute of the `saAmfSG` object class (see Section 8.9)
  Default value: no limit, a value of 0 is used to specify this.
- **Auto-adjust option**: for the general explanation of this option, refer to Section 3.6.1.1 on page 110.
  Section 3.6.4.6 on page 158 shows an example for handling the auto-adjust option in this redundancy model.
  Default value: no auto-adjust

3.6.4.4 SI Assignments

In this section, the general direction in assigning SIs to service units is discussed. Then, a few examples will be given for illustration.

If available service units in the cluster allow it, the Availability Management Framework will instantiate the preferred number of in-service service units for the service group. Then, the preferred number of assigned service units will be used for SI assignments. The remaining in-service service units, if any, will be spare.

It is assumed that the service group configuration has passed a series of validations, so that when as many service units as the preferred number of assigned service units have been assigned, for each configured SI in the service group, a service unit is

assigned active for this SI, and the preferred number of standby assignments is ensured without violating the limits expressed in Section 3.6.4.3.

The following example of a service group configuration will be used throughout this section:

- Ordered list of service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- Ordered list of SIs = {SI1, SI2, SI3, SI4, SI5, SI6}
- Ranked service units for SI1 = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- Preferred number of standby assignments for SI1 = 5
- Ranked service units for SI2 = {SU2, SU3, SU4, SU5, SU6, SU7, SU8, SU1}
- Preferred number of standby assignments for SI2 = 5
- Ranked service units for SI3 = {SU3, SU4, SU5, SU6, SU7, SU8, SU1, SU2}
- Preferred number of standby assignments for SI3 = 5
- Ranked service units for SI4 = {SU4, SU5, SU6, SU7, SU8, SU1, SU2, SU3}
- Preferred number of standby assignments for SI4 = 5
- Ranked service units for SI5 = {SU5, SU6, SU7, SU8, SU1, SU2, SU3, SU4}
- Preferred number of standby assignments for SI5 = 5
- Ranked service units for SI6 = {SU6, SU7, SU8, SU1, SU2, SU3, SU4, SU5}
- Preferred number of standby assignments for SI6 = 5
- Preferred number of in-service service units = 8
- Preferred number of assigned service units = 7
- Maximum number of active SIs per service unit = 3
- Maximum number of standby SIs per service unit = 5

Assignment I: Full Assignment with Spare Service Units

Assume that under the current state of the cluster, all service units can be brought in-service. Then, a running configuration for the service group can be as follows:

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7}
- spare service units = {SU8}

Then, the assignments look like:

- SI1's assignments = {active: SU1; standby: SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {active: SU2; standby: SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {active: SU3; standby: SU4, SU5, SU6, SU7, SU1}
- SI4's assignments = {active: SU4; standby: SU5, SU6, SU7, SU1, SU2}
- SI5's assignments = {active: SU5; standby: SU6, SU7, SU1, SU2, SU3}
- SI6's assignments = {active: SU6; standby: SU7, SU1, SU2, SU3, SU4}

The following points should be mentioned regarding the preceding assignments:

(1) The selection of instantiated service units is based on the ordered list of service units.

(2) The assignments of SIs to service units is based on the ordered list of service units for each SI.

3.6.4.4.1 Reduction Procedure

The difficulty comes when the number of in-service service units is not enough to satisfy the configuration requirements listed in the example. The first goal is to try to keep all SIs in the wanted redundant mode (that is, one service unit is assigned active for each of these SIs, and the preferred number of standby assignments is ensured), even at the expense of imposing maximum load on each service unit. If this goal is not attainable, the next goal is to make sure that as many SIs as possible have active assignments. This may mean a reduction in the number of standby assignments. The reduction is done for less important SIs first. Finally, if this objective is also not attainable, the only choice is to drop some of the SIs completely.

Because the reduction algorithm is simple and somehow similar to the reduction procedure discussed in the N+M case, the reduction procedure is not discussed, and only examples are given.

Assignment II: Full Assignment with Spare Reduction

Assume that initially the service units that can be brought in-service are SU1, SU2, SU3, SU4, SU5, SU6, and SU7. Then, the following can be a running configuration for the service group.

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7}
- spare service units = {}

Then, the assignments look like:

- SI1's assignments = {active: SU1; standby: SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {active: SU2; standby: SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {active: SU3; standby: SU4, SU5, SU6, SU7, SU1}
- SI4's assignments = {active: SU4; standby: SU5, SU6, SU7, SU1, SU2}
- SI5's assignments = {active: SU5; standby: SU6, SU7, SU1, SU2, SU3}
- SI6's assignments = {active: SU6; standby: SU7, SU1, SU2, SU3, SU4}

Assignment III: Full Assignment with Reduction of Assigned Service Units

Assume that the state of the cluster is initially such that only SU1, SU2, SU3, SU4, SU5, SU6 can be brought in-service. Then, the state of the service units is:

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6}
- spare service units = {}

Then, the assignments look like:

- SI1's assignments = {active: SU1; standby: SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {active: SU2; standby: SU3, SU4, SU5, SU6, SU1}
- SI3's assignments = {active: SU3; standby: SU4, SU5, SU6, SU1, SU2}
- SI4's assignments = {active: SU4; standby: SU5, SU6, SU1, SU2, SU3}
- SI5's assignments = {active: SU5; standby: SU6, SU1, SU2, SU3, SU4}
- SI6's assignments = {active: SU6; standby: SU1, SU2, SU3, SU4, SU5}

Assignment IV: Partial Assignment with Reduction of SIs Redundancy Level

Assume that the state of the cluster is such that only the following service units can be brought in-service:

- in-service service units = {SU1, SU2, SU3}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {}

Then, the assignments look like:

- SI1's assignments = {active: SU1; standby: SU2, SU3}
- SI2's assignments = {active: SU2; standby: SU3, SU1}
- SI3's assignments = {active: SU3; standby: SU1, SU2}
- SI4's assignments = {active: SU1; standby: SU2, SU3}
- SI5's assignments = {active: SU1; standby: SU2, SU3}
- SI6's assignments = {active: SU2; standby: SU3, SU1}

Assignment V: Partial Assignment with SIs Drop-Outs

Assume that the state of the cluster is such that only SU1 can be brought in-service. Then, the cluster status looks like:

- in-service service units = {SU1}
- instantiable service units = {}
- assigned service units = {SU1}
- spare service units = {}

- SI1's assignments = {active: SU1; standby: none}
- SI2's assignments = {active: SU1; standby: none}
- SI3's assignments = {active: SU1; standby: none}
- SI4's assignments = {active: none; standby: none}
- SI5's assignments = {active: none; standby: none}
- SI6's assignments = {active: none; standby: none}

3.6.4.5 Failure Handling

In this section, the fail-over action initiated by a node failure is described. Assume that the node hosting SU3 fails. The assignments before the node hosting SU3 failed and after the fail-over completion are as follows:

Assignments Before the Node Hosting SU3 Fails

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6}
- spare service units = {}

- SI1's assignments = {active: SU1; standby: SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {active: SU2; standby: SU3, SU4, SU5, SU6, SU1}
- SI3's assignments = {active: SU3; standby: SU4, SU5, SU6, SU1, SU2}
- SI4's assignments = {active: SU4; standby: SU5, SU6, SU1, SU2, SU3}
- SI5's assignments = {active: SU5; standby: SU6, SU1, SU2, SU3, SU4}
- SI6's assignments = {active: SU6; standby: SU1, SU2, SU3, SU4, SU5}

Assignments After Completion of the Fail-Over

- in-service service units = {SU1, SU2, SU4, SU5, SU6}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU4, SU5, SU6}
- spare service units = {}

- SI1's assignments = {active: SU1; standby: SU2, SU4, SU5, SU6}
- SI2's assignments = {active: SU2; standby: SU4, SU5, SU6, SU1}
- SI3's assignments = {active: SU4; standby: SU5, SU6, SU1, SU2}
- SI4's assignments = {active: SU4; standby: SU5, SU6, SU1, SU2}
- SI5's assignments = {active: SU5; standby: SU6, SU1, SU2, SU4}
- SI6's assignments = {active: SU6; standby: SU1, SU2, SU4, SU5}

When the node hosting SU3 fails, the Availability Management Framework makes adjustments by removing assignments of the SIs from SU3. In this example, it is assumed that the ordering of standby assignments is important. This means that the Availability Management Framework has to inform the components of some service units of the change in their active/standby HA states. For instance, in this example, the Availability Management Framework should do the following for SI1:

- Ask the components of SU4 to go to standby-level 2 for SI1 (it was standby-level 3 before).
- Ask the components of SU5 to go to standby-level 3 for SI1 (it was standby-level 4 before).
- Ask the components of SU6 to go to standby-level 4 for SI1 (it was standby-level 5 before).

3.6.4.6 Example of Auto-Adjust

The auto-adjust option indicates that it is required that the current (running) configuration of the service group returns to the preferred configuration in which the service instance with highest ranks are active and the highest-ranked SIs are assigned in redundant mode. It is up to the Availability Management Framework implementation to decide when and how the auto-adjust will be initiated. The following example is given for illustration purposes.

Assume that the running configuration of the service group is as follows.

- in-service service units = {SU1, SU2, SU3}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {}

- SI1's assignments = {active: SU1; standby: SU2, SU3}
- SI2's assignments = {active: SU2; standby: SU3, SU1}
- SI3's assignments = {active: SU3; standby: SU1, SU2}
- SI4's assignments = {active: SU1; standby: SU2, SU3}
- SI5's assignments = {active: SU1; standby: SU2, SU3}
- SI6's assignments = {active: SU2; standby: SU3, SU1)

Now, assume that the node hosting SU4 joins the cluster. As a result, SU4 becomes instantiable. It is obvious that this configuration is not the preferred one. If the auto-adjust is initiated (in an implementation-dependent way), and assuming that SU4 could be brought in-service, then the service group configuration is as follows after completion of the auto-adjust procedure:

- in-service service units = {SU1, SU2, SU3, SU4}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4}
- spare service units = {}

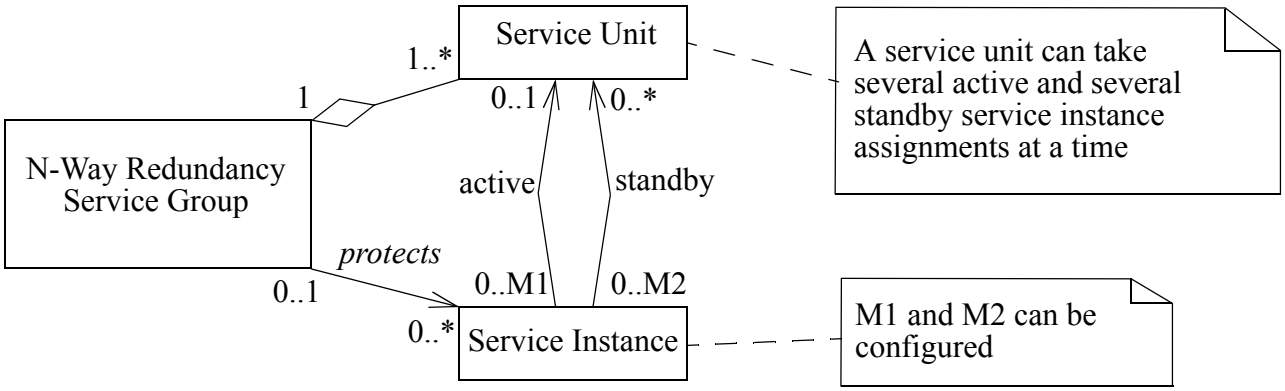
- SI1's assignments = {active: SU1; standby: SU2, SU3, SU4}
- SI2's assignments = {active: SU2; standby: SU3, SU4, SU1}
- SI3's assignments = {active: SU3; standby: SU4, SU1, SU2}

- SI4's assignments = {active: SU4; standby: SU1, SU2, SU3}
- SI5's assignments = {active: SU1; standby: SU2, SU3, SU4}
- SI6's assignments = {active: SU2; standby: SU3, SU4, SU1}

3.6.4.7 UML Diagram of the N-Way Redundancy Model

The N-way redundancy model is represented by the UML diagram shown FIGURE 19.

FIGURE 19 UML Diagram of the N-Way Redundancy Model



3.6.5 N-Way Active Redundancy Model

3.6.5.1 Basics

The **N-way active redundancy model** differs from the 2N, N+M, and N-way redundancy models, as it does not support standby service instance assignments, but allows a service instance to be assigned active to several service units.

For the sake of simplicity, in the subsequent discussion of this redundancy model, it is assumed that the HA readiness state of all service units for all SIs is ready-for-assignment and that the resource capacity limits of the nodes are never exceeded, as the Availability Management Framework assigns and reassigns the SIs.

The characteristics of this redundancy model are:

- Each service unit has to be active for all the SIs assigned to it.
- A service unit is never assigned the standby state for any SI.
- For each SI, none, one, or multiple service units can be assigned the active HA state for that SI.
- The preferred number of active assignments for an SI is an SI-level configuration parameter (see Section 3.6.5.3 on page 162). The preferred number of active assignments may be different for each SI.
- At any given time, several service units can be in-service for a service group: some have SIs assigned to them, and possibly some others are considered spare service units for the service group. The number of assigned service units and the number of spare service units are dynamic and can change during the life-span of the service group; however, the preferred number of these service units can be configured.
- At any given time, the Availability Management Framework should make sure that the redundancy level (the preferred number of active assignments) for each SI is guaranteed (if possible) while the maximum number of SIs assigned to each service units is not exceeded.
- Each SI has an ordered list of service units to which the SI can be assigned. The ordered list of service units per SI must include all the service units configured for the service group. In other words, a partial list of service units is an invalid configuration. If the number of instantiated service units allows it, the Availability Management Framework should make sure that the highest-ranked available service units are assigned active for the SI.

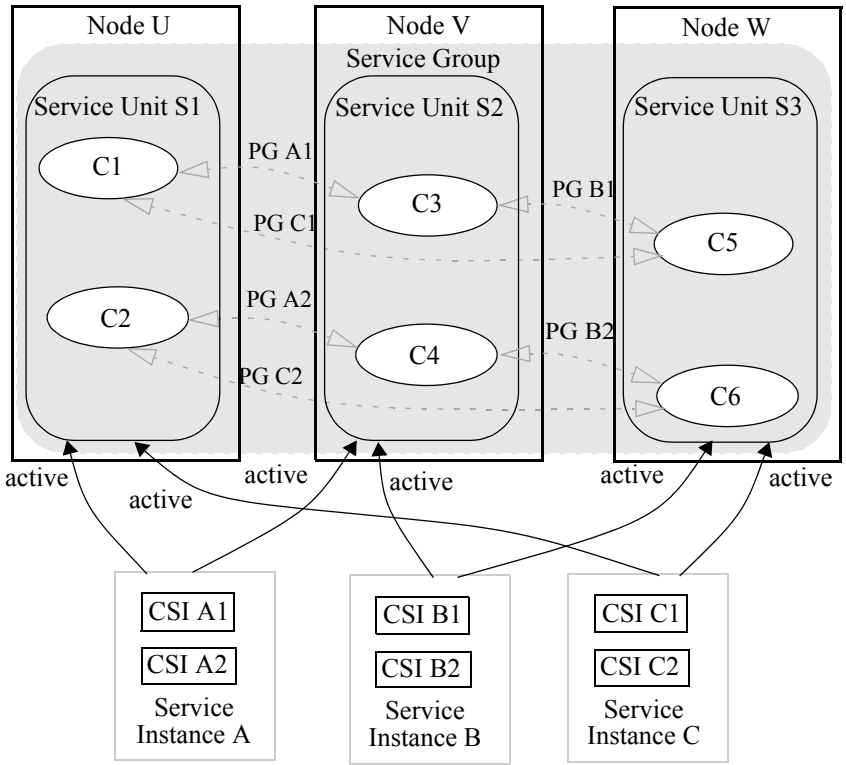
The simplest case for the N-way active redundancy model is the 2-way active redundancy model in which the service group contains two service units that are both assigned the active HA state for every service instance that they support. This configuration is sometimes referred to as an **active-active redundancy configuration**.

Components implementing any of the capability models described in Section 3.5 on page 107 can participate in the N-way active redundancy model.

3.6.5.2 Example

FIGURE 20 next shows an example of the N-way active redundancy model. Note that the HA state of each component for all component service instances assigned to it is active.

FIGURE 20 Example of the N-Way Active Redundancy Model



3.6.5.3 Configuration

- **Ordered list of service units for a service group**: this parameter is described in Section 3.6.1.1.
  Default value: no default, the order is implementation-dependent.
- **Ordered list of SIs**: for the general meaning of this parameter, refer to its definition in Section 3.6.1.1. The Availability Management Framework uses this ranking to choose the SIs with less redundancy (that is, the number of service units having the active HA state for them is less than the preferred number of active service units) or to drop them completely if the number of available service units is not enough for a full support of all SIs.
  Default value: no default, the order is implementation-dependent.
- **Ranked service unit list per SI**: each SI has an ordered list of service units to which the SI can be assigned. This list must be an ordered list consisting of all service units configured for the service group. The rank of a service unit for an SI is configured by setting the `saAmfRank` attribute of a service unit identified by `safRankedSu` in the `SaAmfSIRankedSU` association class (see Section 8.11). The rank is represented by a positive integer. The lower the integer value, the higher the rank.
  The Availability Management Framework should make sure that the highest-ranked available service unit be assigned active for the SI, if possible.
  Default value: the ordered service units list defined for the service group.
- **Preferred number of active assignments per SI**: this parameter indicates the preferred number of service units being assigned the active HA state for each SI. This preferred number is configured by setting the `saAmfSIPrefActiveAssignments` attribute of the `saAmfSI` object class (see Section 8.11).
  Default value: the preferred number of assigned service units.
- **Preferred number of in-service service units**: the Availability Management Framework should make sure that this number of service units are always instantiated, if possible. This preferred number is configured by setting the `saAmfSGNumPrefInserviceSUs` attribute of the `saAmfSG` object class (see Section 8.9).
  Default value: the number of the service units configured for the service group.
- **Preferred number of assigned service units**: this parameter indicates the preferred number of assigned service units at any time. This preferred number is configured by setting the `saAmfSGNumPrefAssignedSUs` attribute of the `saAmfSG` object class (see Section 8.9). As to be discussed later, the Availability Management Framework should guarantee that this number of assigned service units exist for the service group provided that the number of instantiated service units is large enough.
  Default value: the preferred number of in-service service units.

- **Maximum number of active SIs per service unit**: this parameter indicates the maximum number of SIs that can be concurrently assigned to a service unit, so that the service unit has the active HA state for all these SIs. It is assumed that the load imposed by each SI is the same. This maximum number is configured by setting the `saAmfSGMaxActiveSIsperSU` attribute of the `saAmfSG` object class (see Section 8.9).
  Default value: no limit, a value of 0 is used to specify this.
- **Auto-adjust option**: for the general explanation of this option, refer to Section 3.6.1.1 on page 110.
  Section 3.6.5.6 on page 172 shows an example for handling the auto-adjust option in this redundancy model.
  Default value: no auto-adjust

3.6.5.4 SI Assignments

First, the general direction in assigning SIs to service units is discussed. Then, a few examples will be given for illustration.

If the number of available service units in the cluster allows it, the Availability Management Framework will instantiate the preferred number of in-service service units for the service group. Then, the preferred number of in-service service units will be assigned the active HA state for each SI. The remaining instantiated service units will be spare if allowed by the configuration. It is assumed that the service group configuration has passed a series of validations, so that when as many as the preferred number of assigned service units have been assigned, all SIs configured for the service group are assignable, and each SI will have the preferred number of active assignments without violating the limits expressed in the configuration section.

The following example of a service group configuration will be used throughout this section.

- Ordered list of service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8, SU9}
- Ordered list of SIs = {SI1, SI2, SI3, SI4, SI5, SI6}
- Ranked service units for SI1 = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8, SU9}
- Preferred number of active assignments for SI1 = 6
- Ranked service units for SI2 = {SU2, SU3, SU4, SU5, SU6, SU7, SU8, SU9, SU1}
- Preferred number of active assignments for SI2 = 6

- Ranked service units for SI3 = {SU3, SU4, SU5, SU6, SU7, SU8, SU9, SU1, SU2}
- Preferred number of active assignments for SI3 = 6
- Ranked service units for SI4 = {SU4, SU5, SU6, SU7, SU8, SU9, SU1, SU2, SU3}
- Preferred number of active assignments for SI4 = 6
- Ranked service units for SI5 = {SU5, SU6, SU7, SU8, SU9, SU1, SU2, SU3, SU4}
- Preferred number of active assignments for SI5 = 6
- Ranked service units for SI6 = {SU6, SU7, SU8, SU9, SU1, SU2, SU3, SU4, SU5}
- Preferred number of active assignments for SI6 = 6
- Preferred number of in-service service units = 9
- Preferred number of assigned service units = 8
- Maximum number of active SIs per service unit = 5

Assignment I: Full Assignment with Spare

Assume that under the current state of the cluster, all service units can be brought in-service. Then, the following can be a running configuration for the service group.

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8, SU9}
- instantiable service units ={}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- spare service units = {SU9}

Then, the assignments look like:

- SI1's assignments = {SU1, SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU3, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU3, SU4}

The following points should be mentioned regarding the preceding assignments:

(1) The selection of in-service service units is based on the ordered list of service units.

(2) The assignments of SIs to service units are based on the ordered list of service units for each SI.

3.6.5.4.1 Reduction Procedure

The difficulty comes when the number of in-service service units is not enough to satisfy the requirements listed in the configuration. The first goal is to try to keep all SIs in the preferred redundancy levels (that is, with the preferred number of active assignments), even at the expense of imposing maximum load on each service unit. If this goal is not attainable, the next goal is to keep as many important SIs as possible in the preferred redundancy levels without dropping any SIs completely. This may mean reducing the number of assignments for some SIs. The reduction is done for less important SIs first. Finally, if this objective is also not attainable, the only choice is to drop some of the SIs completely (starting first with least important service units).

Because the reduction algorithm is simple and somehow similar to the reduction procedures discussed in the N+M and N-way cases, the reduction procedure is not discussed, and only examples are given.

Assignment II: Full Assignment with Spare Reduction

Assume that under the current state of the cluster, SU9 cannot be instantiated. Then, the following can be a running configuration for the service group.

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

The assignments look like:

- SI1's assignments = {SU1, SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU3, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU3, SU4}

Assignment III: Full Assignment with Maximum Assignments per Service Unit

The reduction procedure should first attempt to keep full assignments (that is, all SIs being supported at their preferred number of active assignments) by loading the service units as much as possible. This first step in the procedure can succeed only if the following condition is fulfilled:

(Maximum number of assignments that can be supported by all in-service service units)

>=

(Number of assignments needed for all SIs given the preferred number of active assignments)

AND

(Number of in-service service units) >= (Maximum of all preferred number of assignments for SIs).

This means that for the example configuration, full assignment is possible only if more than seven service units are instantiated. In the previous example, full assignment is not possible if one of the service units becomes unavailable.

Assignment IV: Partial Assignment with Reduction of SIs Redundancy Level

Assume that the state of the cluster is such that only SU1, SU2, and SU3 can be instantiated:

- in-service service units = {SU1, SU2, SU3}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {}

Then, the assignments look like:

- SI1's assignments = {SU1, SU2, SU3}
- SI2's assignments = {SU2, SU3, SU1}
- SI3's assignments = {SU3, SU1, SU2}
- SI4's assignments = {SU1, SU2, SU3}
- SI5's assignments = {SU1, SU2}
- SI6's assignments = {SU3}

Note that the number of assignments for SIs is reduced to cope with the shortage of in-service service units. The basic logic for assigning SIs to service units can be summarized as follows.

The number of assignments that can be handled in this case is
number of in-service service units (that is, 3) * maximum number of SIs per service unit (that is, 5).
This means that in this example all available in-service service units can handle 15 SI assignments. This may force the Availability Management Framework to decide that the four most important SIs (that is, SI1, SI2, SI3, and SI4) will have three assignments, SI5 two assignments, and SI6 one assignment, as shown above.

Assignment V: Partial Assignment with SIs Drop-Outs

Assume that the state of the cluster is such that only SU1 can be instantiated:

- in-service service units = {SU1}
- instantiable service units = {}
- assigned service units = {SU1}
- spare service units = {}

- SI1's assignments = {SU1}
- SI2's assignments = {SU1}
- SI3's assignments = {SU1}
- SI4's assignments = {SU1}
- SI5's assignments = {SU1}
- SI6's assignments = {}

Note that it was impossible to keep assignments for all SIs in this example, so that the least important SI, SI6, was dropped.

3.6.5.5 Failure Handling

The failure recovery is required to avoid one or both of the following undesirable situations after the occurrence of a failure:

(a) Some of the in-service service units have additional capacity to support more SIs, while some SIs are not being supported with their preferred number of active assignments. In this case, the Availability Management Framework should fill the slack capacity by assigning more service units active for these SIs.

(b) Some less important SIs have more active assignments than those for some more important SIs. In this case, the Availability Management Framework should rearrange SI assignments such that more important SIs get assigned, if possible. This, of course, may require removing some assignments of less important SIs.

The following subsection provides example for the cases (a) and (b):

3.6.5.5.1 Example for Failure Recovery

In this example, assume that the node hosting SU3 fails.

Assignments Before the Node Hosting SU3 Fails

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

Then, the assignments look like:

- SI1's assignments = {SU1, SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU3, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU3, SU4}

Assignments After Failure of the Node Hosting SU3, and Before the Recovery

- in-service service units = {SU1, SU2, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

Then, the assignments look like:

- SI1's assignments = {SU1, SU2, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU4}

In this case, the number of assignments for SIs look like:

- Number of current assignments for SI1 = 5
- Number of current assignments for SI2 = 5
- Number of current assignments for SI3 = 5
- Number of current assignments for SI4 = 6
- Number of current assignments for SI5 = 6
- Number of current assignments for SI6 = 5

The number of SIs assigned to service units is:

- Number of assignments on SU1 = 4
- Number of assignments on SU2 = 4
- Number of assignments on SU4 = 5
- Number of assignments on SU5 = 5
- Number of assignments on SU6 = 5
- Number of assignments on SU7 = 5
- Number of assignments on SU8 = 4

This result is not “optimal” for the following two reasons:

(1) the less important SIs (that is, SI4 and SI5) have higher levels of assignment than more important SIs (that is, SI1, SI2, and SI3);

(2) some in-service service units (that is, SU1, SU2, and SU8) have free capacity while some SIs are not assigned to as many service units as the preferred number of assigned service units.

This situation requires failure recovery, which is discussed next.

Assignments After Completion of Failure Recovery

The failure recovery procedure is implementation-dependent, but the Availability Management Framework implementation should have the ultimate goal of maximizing the number of active assignments for the most important SIs (obviously, this number may not be higher than the preferred number of active assignments per SI); however, to attain this goal, the Availability Management Framework implementation may require complex reassignment algorithms; therefore, the specification does not enforce this goal to the implementation. At the end of this subsection, a more practical (but less ambitious) goal for failure recovery is given.

Because the overall capacity of the service units is 35 (7 SIs with 5 assignments each), SI1 through SI5 should get full assignments and only SI6 should get partial assignments. According to this objective, the following can be the post-recovery assignments:

- in-service service units = {SU1, SU2, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

- SI1's assignments = {SU1, SU2, SU8, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU1, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU2, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU4}

This means the following additional assignments:

- SI1 assigned to SU8
- SI2 assigned to SU1
- SI3 assigned to SU2

These assignments guarantee that the most important SIs get the highest number of assignments possible under the existing configuration limitations (hence, it is called an optimal assignment).
As noted earlier, the failure recovery procedure is implementation-dependent. Thus, some simpler implementations may not arrive at the mentioned “optimal” solution. For example, a simple implementation that does not aim to guarantee "highest possible assignments to the most important SIs", but attempts to adjust the assignments partially (without service group level optimization), may end up with the following post-recovery configuration:

- SI1's assignments = {SU1, SU2, SU7, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU1, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU2, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU8, SU1, SU2, SU4}

In this example, each of the SIs affected by the service unit failure is assigned to another service unit. For example, SI1 is assigned to SU7 as a replacement of its assignment to SU3.

As mentioned at the beginning of this subsection, to make the Availability Management Framework’s implementation simpler, the specification does not require the optimal error recovery (as defined earlier in this section). It only requires that the error recovery procedure achieves the following non-optimal goals:

(a) The more important SIs should get more assignments than less important SIs after the completion of the recovery.

(b) The implementation should minimize the number of SI reassignments during the recovery process.

(c) The free capacity of service units should be kept as small as possible.

3.6.5.6 Example of Auto-Adjust

As discussed earlier, the failure recovery should avoid undesirable situations (that is, underutilized service units and more important SIs not being assigned in higher number); however, the failure recovery may not consider the service units ordered list for assigning SIs.

Thus, in some cases, the SIs are not arranged based on their service units ordered lists. The switch-over procedure can be initiated to do one of the following rearrangements:

(1) redistribute the SIs to service units evenly in accordance with the per-SI ordering, so that the SIs are distributed among all assigned service units;

(2) rearrange the assignment, so that the order of the per-SI service units is honored.

The following example illustrates the auto-adjust procedure.

Assignments Before the Node hosting SU3 Joins

- in-service service units = {SU1, SU2, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

The assignments look like:

- SI1's assignments = {SU1, SU2, SU8, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU1, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU2, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU4}

Now, assume that the node hosting SU3 joins the cluster.

The following will be the service group configuration after the failure recovery.

Assignments After the Node Hosting SU3 Joins

Because only SI6 is not supported with full 6 active assignments, the Availability Management Framework can (at least) assign SU3 active for SI6. Therefore, the following can be the assignments after the node hosting SU3 joins the cluster:

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

After the node hosting SU3 joins, the assignments look like:

- SI1's assignments = {SU1, SU2, SU8, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU1, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU2, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU4, SU3}

If the administrator requests an auto-adjust, the assignments will look like after the completion of the auto-adjust:

Assignments After Completion of the Auto-adjust Procedure

- in-service service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4, SU5, SU6, SU7, SU8}
- spare service units = {}

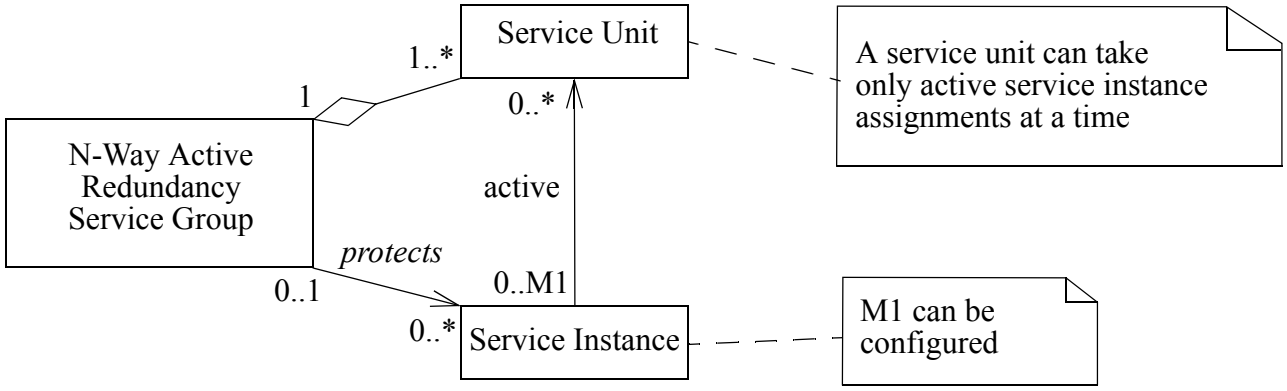
The assignments look like:

- SI1's assignments = {SU1, SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU3, SU4, SU5, SU6, SU7, SU8}
- SI4's assignments = {SU4, SU5, SU6, SU7, SU8, SU1}
- SI5's assignments = {SU5, SU6, SU7, SU8, SU1, SU2}
- SI6's assignments = {SU7, SU8, SU1, SU2, SU3, SU4}

3.6.5.7 UML Diagram of the N-Way Active Redundancy Model

The N-way active redundancy model is represented by the UML diagram shown in FIGURE 21.

FIGURE 21 UML Diagram of the N-Way Active Redundancy Model



3.6.6 No-Redundancy Redundancy Model

3.6.6.1 Basics

The **no-redundancy model** is a very simple model in which a service unit can have at most one service instance assigned and where a service instance is assigned to at most one service unit.

For the sake of simplicity, in the subsequent discussion of this redundancy model, it is assumed that the HA readiness state of all service units for all SIs is ready-for-assignment and that the resource capacity limits of the nodes are never exceeded, as the Availability Management Framework assigns and reassigns the SIs.

This redundancy model is typically used with non-critical components, when the failure of a component does not cause any severe impact on the overall system.

This redundancy model has the following characteristics:

- A service unit is assigned the active HA state for at most one SI. In other words, no service unit will have more than one SI assigned to it.
- A service unit is never assigned the standby HA state for an SI. The Availability Management Framework can recover from a fault only by failing over the active assignment to a spare service unit, if it is available, by restarting a service unit, or—as an escalation—by restarting the node (see Section 9.4.7 on page 383) containing the service unit.
- No two service units exist having the same SI assigned to them.
- At any given time, several in-service service units can be instantiated for a service group: some have SIs assigned to them, and possibly some others are considered spare service units for the service group. The number of service units that have SIs assigned to them and the number of spare service units are dynamic and can change during the life-span of the service group; however, the preferred number of in-service service units can be configured.
- At any given time, the Availability Management Framework should ensure that each SI is assigned to a service unit provided that the number of in-service service units is large enough.
- SIs are ordered based on their importance. This ordered list will be used for assigning SIs to service units.

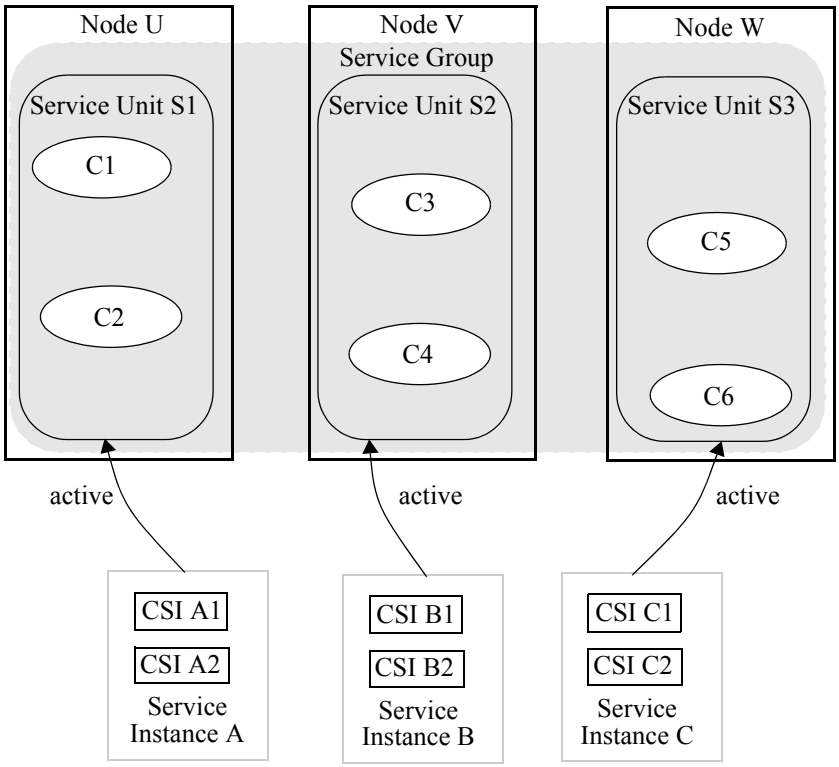
Note: As stated in Section 3.1.6, any service unit in a service group is capable of providing all SIs defined in the configuration for the service group.

Components implementing the x_active_and_y_standby, x_active_or_y_standby, 1_active_or_y_standby, 1_active_or_1_standby, x_active, 1_active, or non-pre-instantiable capability models can participate in the no-redundancy model.

3.6.6.2 Example

An example of the no-redundancy model is shown in FIGURE 22.

FIGURE 22 Example of the No-Redundancy Redundancy Model



3.6.6.3 Configuration

- **Ordered list of service units for a service group**: this parameter is described in Section 3.6.1.1.
  Default value: no default, the order is implementation-dependent.
- **Ordered list of SIs**: for the general meaning of this parameter, refer to its definition in Section 3.6.1.1. The Availability Management Framework uses this ranking to select the SIs to drop from assignment if the number of service units is not enough for a full support of all SIs.
  Default value: no default, the order is implementation-dependent.
- **Preferred number of in-service service units**: the Availability Management Framework should make sure that this number of in-service service units is always instantiated, if possible. This preferred number is configured by setting the `saAmfSGNumPrefInserviceSUs` attribute of the `saAmfSG` object class (see Section 8.9).
  Default value: the number of the service units configured for the service group.
- **Auto-adjust option**: for the general explanation of this option, refer to Section 3.6.1.1 on page 110.
  Section 3.6.6.6 on page 180 shows an example for handling the auto-adjust option in this redundancy model.
  Default value: no auto-adjust

3.6.6.4 SI Assignments

First, the general approach for assigning SIs to service units is discussed. Then, a few examples will be given for illustration.

If the number of available service units in the cluster allows it, the Availability Management Framework will instantiate the preferred number of instantiated service units for the service group. Then, some or all of these service units will be used for SI assignments. The remaining instantiated service units will be spare. It is assumed that the service group configuration has passed a series of validations, so that when the required number of service units is assigned, each configured SI can be assigned to a service unit.

The following example of a service group configuration will be used throughout this section.

- Ordered list of service units = {SU1, SU2, SU3, SU4, SU5}
- Ordered list of SIs = {SI1, SI2, SI3}
- Preferred number of in-service service units = 4

Assignment I: Full Assignment with Spare

Assume that under the current state of the cluster, all service units can be brought in-service. Then, the following can be a running configuration for the service group.

- in-service service units = {SU1, SU2, SU3, SU4}
- instantiable service units = {SU5}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {SU4}

Then, the assignments looks like:

- SI1's assignment = {SU1}
- SI2's assignment = {SU2}
- SI3's assignment = {SU3}

The following points should be mentioned regarding these assignments:

(1) The selection of in-service service units is based on the ordered list of service units.

(2) The assignments of SIs to service units are based on the ranking of the SIs in the ordered list of SIs.

3.6.6.4.1 Reduction Procedure

The first goal of the assignment procedure is to try to keep all SIs assigned. If this goal is not attainable, then the next goal is to keep as many important SIs as possible assigned.

Assignment II: Full Assignment with Spare Reduction

Assume that under the current state of the cluster, SU4 and SU5 cannot be instantiated. Then, the following can be a running configuration for the service group.

- in-service service units = {SU1, SU2, SU3}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {}

Then, the assignments looks like:

- SI1's assignment = {SU1}
- SI2's assignment = {SU2}
- SI3's assignment = {SU3}

Assignment III: Partial Assignment

If the number of instantiated service units is not large enough, some less important SIs will be dropped. Assume that only SU1 and SU2 can be brought in-service in this example.

- in-service service units = {SU1, SU2}
- instantiable service units = {}
- assigned service units = {SU1, SU2}
- spare service units = {}

Then, the assignments look like:

- SI1's assignment = {SU1}
- SI2's assignment = {SU2}
- SI3's assignment = {}

3.6.6.5 Failure Handling

The failure handling is rather simple. If a node hosting a service unit fails, the only fail-over option is to select a spare service unit from the service group's spare service units and to assign the SI of the failed service unit to the selected spare service unit. If no spare service unit is available, the Availability Management Framework cannot carry out any failure handling, and the SI that was being provided by the failed service unit will not be supported until another service unit becomes available for the service group.

The following example illustrates the fail-over action.

Assignments Before the Node Hosting SU3 Failed

- in-service service units = {SU1, SU2, SU3, SU4}
- instantiable service units = {SU5}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {SU4}

- SI1's assignments = {SU1}
- SI2's assignments = {SU2}
- SI3's assignments = {SU3}

Assignments After the Failure Recovery

- in-service service units = {SU1, SU2, SU4, SU5}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU4}
- spare service units = {SU5}
- SI1's assignments = {SU1}
- SI2's assignments = {SU2}
- SI3's assignments = {SU4}

3.6.6.6 Example of Auto-Adjust

The auto-adjust procedure does not achieve much in this redundancy model. It only makes sure that the SIs are assigned to the most preferred in-service service units. The following example illustrates the auto-adjust procedure.

Assignments Before the Auto-adjust Procedure

After the node hosting SU3 joins the cluster (see previous example), the service units and the assignments can be as follows:

- in-service service units = {SU1, SU2, SU4, SU5}
- instantiable service units = {SU3}
- assigned service units = {SU1, SU2, SU4}
- spare service units = {SU5}

- SI1's assignments = {SU1}
- SI2's assignments = {SU2}
- SI3's assignments = {SU4}

Because the ranking of SU3 for SI3 is higher than the ranking of SU4 for SI3, if the auto-adjust option is enabled for the service group when SU3 is brought in-service again, the assignments will look like:

Assignments After the Auto-adjust Procedure

- in-service service units = {SU1, SU2, SU3, SU4}
- instantiable service units = {SU5}
- assigned service units = {SU1, SU2, SU3}
- spare service units = {SU4}

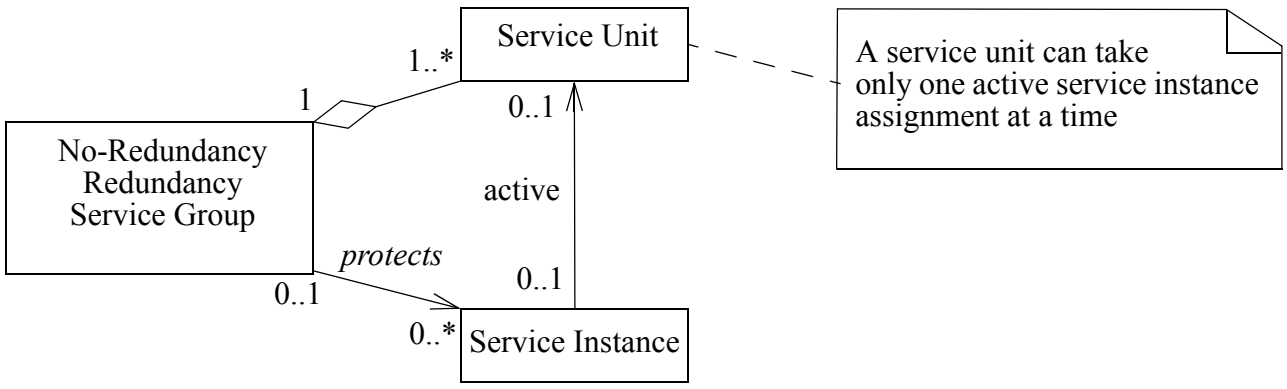
- SI1's assignments = {SU1}
- SI2's assignments = {SU2}
- SI3's assignments = {SU3}

Note that SU5 has been uninstantiated, because the number of preferred service units is 4.

3.6.6.7 UML Diagram of the No-Redundancy Redundancy Model

The no-redundancy redundancy model is represented by the UML diagram shown in FIGURE 23.

FIGURE 23 UML Diagram of the No-Redundancy Redundancy Model



3.6.7 The Effect of Administrative Operations on Service Instance Assignments

Usually, administrative operations such as lock or unlock of a service unit or of a node result in reassignments of SIs to service units. This section briefly discusses how the lock and unlock administrative operations affect SI assignments. The cases for other administrative operations are similar.

Only basic directions are given here, as the detailed reaction of the Availability Management Framework for each administrative operation depends on the redundancy model. The details are left for the implementation.

3.6.7.1 Locking a Service Unit or a Node

As the lock for instantiation does not affect the service instance assignment, this subsection focusses only on the lock administrative operation.

Depending on the status of the service unit, a lock administrative operation affects the SIs assigned to the service unit as follows:

(a) The service unit (say, SU1) or one of its enclosing entities like the node, service group, application, or the cluster is being locked, and the service unit has SI assignments: in this case, the SIs supported by the service units will be reassigned to other service units in the service group. This reassignment depends obviously on the redundancy model of the service group. The transfer of SI assignments from the service unit SU1 to other service units is very similar to the recovery operation performed when a service units fails. For details, refer to the failure handling section of the associated redundancy model. However, it is important to note that an effective reassignment may require selecting one of the spare service units or instantiating a new service unit from the instantiable set. The removal of SI assignments will not trigger a termination of the service unit, unless it is non-pre-instantiable, and the operation discussed in case (b) is undertaken when the service unit enters the out-of-service readiness state.

(b) The service unit (say SU1) or one of its enclosing entities like the node, service group, application, or the cluster is being locked, and the service unit has no current SI assignments, but it belongs to the set of in-service service units: in this case, when the service unit SU1 becomes out-of-service, and, as a consequence, the number of in-service service units drops below the preferred number of in-service service units, one instantiable service unit with none of its containing entities (service group, node, application, or cluster) in locked state will be selected to replace the service unit SU1. This selection will be based on the service units and their ranks, as discussed in Section 3.6.1. The service unit SU1 stays in the set of instantiated service unit.

(c) The service unit to be locked does not belong to the set of in-service service units: in this case, no SI reassignment or service unit instantiation is performed.

3.6.7.2 Unlocking a Service Unit, a Service Group, or a Node

Depending on the status of the service unit, an unlock administrative operation may lead to assignments of SIs to the service unit. Three cases need consideration:

(a) The service unit does not belong to the set of instantiable service units:
in this case, the service unit still remains out of the set of instantiated service units; thus, no SIs can be assigned to the service unit.

(b) The service unit belongs to the set of instantiable service units, but it is not instantiated:
in this case, if the preferred number of in-service service units is not reached, the service unit is instantiated. If the service unit can be brought in-service, the operation described in case (c) is undertaken.

(c) The service unit is in-service:
based on the configuration of the service group (auto-adjust option and preferred number of assignments) and on the current assignments, some SIs may be assigned to the service unit.

3.7 Component Capability Model and Service Group Redundancy Model

A component having a certain component capability model can only participate in a certain set of service group redundancy models. This mapping between the component capability models and the service group redundancy models is shown in Table 16.

Table 16 Component Capability Model and Service Group Redundancy Model

Service Group Redundancy Model —> ---------- Component Capability Model	2N	N+M	N-Way	N-Way Active	No-Redundancy
x_active_and_y_standby	X	X	X	X	X
x_active_or_y_standby	X	X	-	X	X
1_active_or_y_standby	X	X	-	X	X
1_active_or_1_standby	X	X	-	X	X
x_active	X	X	-	X	X
1_active	X	X	-	X	X
non-pre-instantiable component	X	X	-	X	X

A component with capability models x_active or 1_active for a certain component service type is eligible for being used in service groups with redundancy models 2N and N+M. The component may have the active, quiescing, or quiesced HA states, but not the standby HA state for its CSIs. Nevertheless, its service unit can be assigned the standby HA state for a service instance. The Availability Management Framework does not attempt to assign the standby HA state for a CSI to the component in this case.

3.8 Dependencies Among SIs, Component Service Instances, and Components

3.8.1 Dependencies Among Service Instances and Component Service Instances

The Availability Management Framework defines two types of dependencies between service instances (SI) and component service instances (CSI):

- SI—> SI, cluster wide.
- CSI—> CSI in the same SI.

The **SI-SI dependencies** are also applicable to applications (see also Section 3.1.7).

The dependencies apply in two cases.

- When a service unit is assigned the active HA state on behalf of a service instance or a component is assigned the active HA state on behalf of a component service instance.
- When the active HA state was assigned to a service unit or a component on behalf of a service instance or component service instance, respectively, and another HA state is now assigned, or the active HA state assignment is removed.

3.8.1.1 Dependencies Among SIs when Assigning a Service Unit Active for a Service Instance

A service instance SI1 may be configured to depend on other service instances (especially within the scope of an application logical entity, as defined in Section 3.1.7), SI2, SI3, and so on, in the sense that a service unit can only be assigned the active HA state for SI1 if all SI2, SI3, and so on are either fully-assigned or partially-assigned (see Section 3.2.3.2). A dependency between a service instance SI1 on a service instance SI2 is configured by specifying the DN of SI2 in the `safDepend` attribute in an object of the `SaAmfSIDependency` association class that is associated with the object representing SI1 (see Section 8.11).
These dependencies are cluster-wide, which means that the service instances on which a service instance SI1 depends can belong to the same service group as SI1 or to another service group.

The rank of a dependent service instance must not be higher than the ranks of the service instances on which it depends.

3.8.1.2 Impact of Disabling a Service Instance on the Dependent Service Instances

The Availability Management Framework defines one configurable attribute of a dependency between service instances, the **tolerance time**: if a service instance SI1 depends on the service instance SI2, this time indicates for how long SI1 can tolerate SI2 being in the unassigned state (see Section 3.2.3.2). If this time elapses before SI2 becomes assigned again, the Availability Management Framework will remove the active and the quiescing HA states for SI1 from all service units, that is, it will make SI1 unassigned. The tolerance time is configured by setting the `saAmfToleranceTime` attribute of the `SaAmfSIDependency` association class (see Section 8.11). This tolerance time can be set to zero to indicate to the Availability Management Framework that it must remove the active and the quiescing HA states for SI1 from all service units immediately as soon as SI2 is unassigned.

3.8.1.3 Dependencies Among Component Service Instances of the same Service Instance

A component service instance of a service instance can be configured to depend on other component service instances of the same service instance. This **dependency amongst component service instances** is configured by specifying a list of the DNs of the component service instances on which a component service instance depends in the `saAmfCSIDependencies` attribute of the `saAmfCSI` object class (see Section 8.12).
The Availability Management Framework performs the assignment of the active HA state to components on behalf of component service instances in a sequence determined by these dependencies: if a component service instance CSI1 depends on the component service instance CSI2, a component can only be assigned the active HA state for CSI1 if a component of the service unit in question has already acknowledged the assignment of the active HA state for CSI2 by calling the `saAmfResponse_4()` function. The reverse order is applied when, on behalf of component service instances, the active HA state is removed from components or another HA state is assigned to components.

Example: suppose a component `C1` consisting of an HTTP server supporting a component service instance `CSI1` that contains an IP address and a port number. The server binds to that IP address (and not to `INADDR_ANY`) and to that port number. A second component `C2` implements a virtual IP address service, and its component service instance `CSI2` contains simply the same IP address as above. `CSI2` must be assigned before `CSI1`; otherwise the `bind()` system call would fail.

3.8.2 Dependencies Among Components

A component can be configured to depend on another component in the same service unit in the sense that the instantiation of the second component is a prerequisite for the instantiation of the first component.

Dependencies amongst components described in this section are applicable only when instantiating or terminating a service unit. These dependencies in no way influence the state transitions effected by the Availability Management Framework.

Such explicit dependencies can be configured between any two pre-instantiable components in the same service unit. Note that implicit dependencies exist between a proxy and its proxied components—not to be discussed here.

A system administrator can take advantage of dependencies amongst components to avoid launching processes that perform a lengthy initialization concurrently, as this could lead to CPU saturation. A "tempered" launching of these processes could be more adequate.

Dependencies amongst components are configured by associating an **instantiation level** with each pre-instantiable component. The instantiation level is a positive integer configured for such components. The corresponding configuration attribute is the `saAmfCompInstantiationLevel` attribute of the `saAmfComp` object class (see Section 8.13.2).

Within a service unit, the Availability Management Framework instantiates the pre-instantiable components according to the configured instantiation level. All pre-instantiable components with the same instantiation level are instantiated by the Availability Management Framework in parallel. Components of a given level are only instantiated by the Availability Management Framework when all components with a lower instantiation level have successfully completed their instantiation.

Within a service unit, the Availability Management Framework terminates the pre-instantiable components according to the configured instantiation level. All pre-instantiable components with the same instantiation level are terminated by the Availability Management Framework in parallel. Pre-instantiable components of a given level are only terminated by the Availability Management Framework when all pre-instantiable components with a higher instantiation level have been terminated.

As has been said previously, the instantiation level is only applicable during service unit instantiation and termination. As restarting a service unit means terminating the service unit and instantiating it again, the instantiation level also applies when restarting a service unit. If single components within a service unit are restarted, the instan-

tiation level does not cause components with a higher level to be also subject to a restart. The instantiation level is, above all, a means to limit the load on the system during the instantiation process.

Non-pre-instantiable components are only instantiated when they have to provide service (for instance, when the Availability Management Framework assigns them the active HA state for a component service instance).
If dependencies amongst a non-pre-instantiable and another component exist, they should be resolved by using the inter-CSI (CSI–CSI) dependency scheme.

3.9 Approaches for Integrating Legacy Software or Hardware Entities

Non-SA-aware software or hardware entities can be integrated into the Availability Management Framework model in two ways:

- By the use of a **wrapper** to encapsulate the legacy software (hardware) into an SA-aware component. The wrapper consists of one or more processes that link with the Availability Management Framework library and interact with the Availability Management Framework on the one hand and with the legacy software (hardware) on the other hand. The wrapper and the legacy software (hardware) together constitute a single component.
- By the use of a proxy to manage the legacy software (hardware). The legacy software (hardware) can be considered to be a separate component managed by the proxy component.

In general, the proxy/proxied solution is appropriate most when one of the following is true:

- The redundancy model of the proxied entity (the legacy software or hardware) is different from the redundancy model of the proxy entity. The proxy entity usually requires a very simple redundancy model such as 2N, whereas the legacy entity may need a more complex redundancy models such as N+M and N-way active.
- The failure semantics and fault zone of the proxied entities are different from the ones for proxy entities. For example, the proxied entity may be running outside the cluster, whereas the proxy entity has to be located on a node.

3.10 Component Monitoring

Three types of **component monitoring** can be envisaged for a component:

- **Passive Monitoring**: the component is not involved in the monitoring, and mostly operating system features are used to assess the health of a component. These features include monitoring the death of processes that are part of the component (but it could also be extended to also monitor crossing some thresholds in resource usage such as memory usage).
- **External Active Monitoring**: the component does not include any special code to monitor its health, but some entity external to the component (usually called a monitor) assesses the health of the component by submitting some service requests to the component and checking that the service is provided in a timely fashion.
- **Internal Active Monitoring**: the component includes code (often called audits) to monitor its own health and to discover latent faults. Each of these health checks is triggered either by the component itself or by the Availability Management Framework.

These three types of monitoring are in fact complementary. Passive monitoring or external active monitoring do not need modification of the component itself and can be applied to non-SA-aware components.

The Availability Management Framework supports these three types of monitoring.

The passive monitoring of components is covered by the API functions `saAmfPmStart_3()` (refer to Section 7.7.1 on page 278) and `saAmfPmStop()` (refer to Section 7.7.2 on page 280).
External active monitoring is supported with two command line interfaces (CLI), namely the commands `AM_START` (refer to Section 4.9 on page 215), which is used to start a monitoring process for a component and `AM_STOP` (refer to Section 4.10 on page 215), which is used to stop a monitoring process for a component.
Due to the extra load put on the system to run CLI commands (need to spawn a process each time), it is preferable to have long running processes for external active monitors (as opposed to run periodically a monitoring command similarly to what is done for audits).

The internal active monitoring of components is accomplished through the health-check interfaces (refer to Section 7.1.2 on page 232).

3.11 Error Detection, Recovery, Repair, and Escalation Policy

3.11.1 Basic Notions

3.11.1.1 Error Detection

Error detection is the responsibility of all entities in the system. Errors are reported to the Availability Management Framework by invoking the `saAmfComponentErrorReport_4()` API function, described in Section 7.12.1 on page 325. The invoker of this function specifies the recommended recovery action, which can assume the values defined in the `SaAmfRecommendedRecoveryT` enum, described in Section 7.4.7 on page 257. Components play an important part in error detection and should report their own errors or the errors of other components with which they interact. The Availability Management Framework itself also generates error reports on components when it detects errors while interacting with components. For the different cases, refer to Section 3.2.2.2 on page 75.

It is assumed that a reported error does not refer explicitly to a specific component service instance currently assigned to the component. It rather applies to the component as a whole.

3.11.1.2 Restart

The **restart** of a component means any of the following sequences of life cycle operations:

- terminate + instantiate
- cleanup + instantiate
- terminate + cleanup + instantiate

The latter sequence applies if an error occurs during the terminate operation.

Appendix A describes how these operations are implemented for the various types of components.

The Availability Management Framework terminates erroneous components abruptly by executing the appropriate cleanup operation for the component (see Table 37 in Appendix A). Non-erroneous components are terminated gracefully by first attempting to run the corresponding callback or the `TERMINATE` command (see also Table 37 in Appendix A).

During a restart because of a failure, a component remains enabled, and its readiness state may or may not change according to changes in its presence state (as described in Section 3.2.2.1), which in turn determines whether its component service instances must be removed (refer to Section 3.2.2.3).

Restarting a service unit is achieved by the following actions:

- First, all components in the service unit are terminated in the order dictated by their instantiation-levels.
- In a second step, all components in the service unit are instantiated in the order dictated by their instantiation-levels.

During this restart procedure, the components follow their relevant state transition (see Section 3.2.2.1), which affects the presence state of the service unit (see Section 3.2.1.1) and, consequently, its readiness state (see Section 3.2.1.4), which in turn determines the service instance assignments. If a service unit contains only restartable components, that is, the `saAmfCompDisableRestart` configuration attribute of all the components is set to SA_FALSE (see the `SaAmfComp` object class in Section 8.13.2), the service unit remains in the in-service readiness state during the restart. As a consequence, its service instance assignments remain intact.

3.11.1.3 Recovery

Recovery is an automatic action taken by the Availability Management Framework (no human intervention)—after an error occurred to a component—to ensure that all component service instances that were assigned to this component, are reassigned to non-erroneous components. This applies to all component service instances regardless of the HA state of the component for these component service instances.

The recovery actions are described in the following subsections.

In this section and also throughout this document, the values defined in the `SaAmfRecommendedRecoveryT` enum, described in Section 7.4.7 on page 257, will be used to designate the corresponding recovery actions, without necessarily referring to this enum.

Note that if a component fails, just removing component service instances from the component and reassigning the component service instances to it (without restarting the component) is not considered as a valid recovery action.

One of the recovery actions described in the following subsections is configured per component as the default recovery action. The corresponding configuration attribute is `saAmfCompRecoveryOnError`, defined in the `SaAmfComp` object class (see Section 8.13.2).

The Availability Management Framework engages the default recovery action under the following circumstances:

- The error report specifies the value `SA_AMF_NO_RECOMMENDATION`.
- A component does not respond to a callback invoked by the Availability Management Framework within a reasonable period of time.
- A component responds with an error to a callback invoked by the Availability Management Framework on the component.

3.11.1.3.1 Restart Recovery Action

The objective here is to avoid reassigning service instances to different service units. The Availability Management Framework tries to fix the problem by restarting some components and reassigning them all component service instances previously assigned with the same HA state. This may not always be possible, as other events that would prevent the Availability Management Framework from performing such assignments (for example some dependencies may not be satisfied anymore) may have occurred during the recovery. The following levels of restart are provided:

⇒ **Restart the erroneous component**: the erroneous component is abruptly terminated and then instantiated again. The Availability Management Framework attempts to reassign component service instances previously assigned to the components with the same HA state. This action is performed as a consequence of an `SA_AMF_COMPONENT_RESTART` recommended recovery action provided in the error report.

⇒ **Restart all components of the service unit**: all components of the service unit that contains the erroneous component are abruptly terminated and then instantiated again (see Section 3.11.1.2). This action is performed as a consequence of an escalation of an `SA_AMF_COMPONENT_RESTART` recommended recovery action.

⇒ **Restart the associated container and all collocated contained components**: The Availability Management Framework performs the following actions in sequence:

- it abruptly terminates the affected contained component and its collocated contained components;
- it terminates the associated container component;
- it instantiates the container component and attempts to reassign component service instances previously assigned to this container component (including the corresponding container CSIs) with the same HA state;
- it requests the container component to instantiate the associated contained components and attempts to reassign component service instances previously assigned to these contained components with the same HA state.

This action is performed as a consequence of an `SA_AMF_CONTAINER_RESTART` recommended recovery action (requested for a contained component) or as a consequence of an `SA_AMF_COMPONENT_RESTART` recommended recovery action requested for a container component.

The Availability Management Framework must provide the option to disable restart recovery actions for particular components. This option should be used when restarting a component takes too much time, and fail-over is a preferred recovery action. See Section 3.2.2.1 on page 71.

3.11.1.3.2 Fail-Over Recovery Action

Either because the restart recovery action has been disabled in the configuration of a particular component (its `saAmfCompDisableRestart` configuration attribute is set to SA_TRUE, see the `SaAmfComp` object class in Section 8.13.2), or because previous attempts to restart the component failed, or because the error report specified another recommended recovery action, the Availability Management Framework may decide to recover by reassigning service instances to service units other than the one to which they are currently assigned. The different levels of fail-over listed next differ by the scope of the service instances being failed over (some service instances assigned to a service unit, or all service instances assigned to services units of a node) and by how abruptly component service instances are removed from the components to which they are currently assigned (regular HA state management leading to the removal of the component service instance, graceful component termination, abrupt component termination, or abrupt node reboot).

- **Component or Service Unit Fail-Over**
  The Availability Management Framework provides the saAmfSUFailover configuration attribute at the service unit level (see the SaAmfSU object class in Section 8.10) to indicate whether a component fail-over should trigger a fail-over of the entire service unit or of only of the erroneous component. The saAmfSUFailover configuration attribute of a non-pre-instantiable service unit must always be set to SA_TRUE
  By default, the saAmfSUFailover configuration attribute of a service unit is set to SA_TRUE.

  If the service unit is configured to fail over as a single entity (saAmfSUFailover set to SA_TRUE), all other components of the service unit are abruptly terminated, and all service instances assigned to that service unit are failed over; otherwise, only the erroneous component is abruptly terminated, and all component service instances that were assigned to it are failed over. Other components are not terminated, but all service instances that contained one of the failed over component service instances have their remaining component service instances switched over. Switch-over means that component service instances are not abruptly removed from components; the HA state of these components for these component service instances is rather transitioned to the quiesced HA state before being removed.

  The following example helps in clarifying this. Assume a service group having some service units, each comprising 3 components. One of these service units, SU1, contains the C1, C2, and C3 components. Now, assume that SU1 is assigned the active HA state for two service instances, SI1 and SI2. SI1 contains 3 CSIs: CSI11, CSI12, and CSI13, which are assigned to C1, C2, and C3, respectively, and SI2 contains only two component service instances, CSI21 and CSI23, which are assigned to C1 and C3, respectively.

  Assume that C2 fails. C2 is abruptly terminated. As C2 was assigned CSI12, CSI12 is failed over and the other component service instances of SI1 need to be switched over, namely CSI11 and CSI13. However, it is not necessary to switch over SI2, as it has no CSIs assigned to the failed component C2.

  In a 2N or N+M redundancy model, SI2 also needs to be switched over; otherwise, the number of active service units would be higher than what is allowed by the redundancy model. However, in an N-way redundancy model, SI2 could be left assigned to SU1 (if the saAmfSUFailover configuration attribute of the service unit is set to SA_FALSE), and a repair of C2 should be attempted by reinstantiating it. If the attempt to instantiate C2 fails, the service unit becomes disabled, and SI2 must be switched-over; however, if the attempt to instantiate

C2 is successful, SI2 shall remain assigned to SU1, and based on other configuration parameters and N-way redundancy model semantics, even SI1 might get reassigned to SU1.

This action is performed as a consequence of an SA_AMF_COMPONENT_FAILOVER recommended recovery action or of an escalation to it.

- **Node Switch-Over**
  This implies an abrupt termination of the failed component and the fail-over of all component service instances that were assigned to it. Component service instances assigned to other components of the service unit are failed over or switched over depending on the setting of saAmfSUFailover attribute of the service unit.
  All service instances assigned to other service units on the node have their component service instances switched over. Switch-over means that component service instances are not abruptly removed from components; the HA state of these components for these component service instances is rather transitioned to the quiesced HA state before being removed.
  This action is performed as a consequence of an SA_AMF_NODE_SWITCHOVER recommended recovery action.
- **Node Fail-Over**
  This implies an abrupt termination of all local components and the fail-over of all service instances assigned to all service units on a node. This action is performed as a consequence of an SA_AMF_NODE_FAILOVER recommended recovery action, or as the result of a recovery escalation.
- **Node Failfast**
  The Availability Management Framework reboots the node by invoking an administrative operation on the appropriate PLM entity without trying to terminate the components individually. The reboot operation must be carried out in such a way that all local components of the node (including its hardware components) are placed into the uninstantiated presence state, which may require powering-down or resetting some hardware entities. As part of the node failfast operation, a fail-over of the service instances assigned to service units on the node is performed. This action is performed as a consequence of an SA_AMF_NODE_FAILFAST recommended recovery action.

3.11.1.3.3 Application Restart Recovery Action

The application should be completely terminated and then started again by first terminating all of its service units and then starting them again, ensuring that—during the termination phase of the restart procedure—service instances of the application are not reassigned (refer additionally to Section 9.4.7 on page 383). It is important to note that it is not required to preserve the pre-restart service instance assignments to various service units in the application upon re-starting an application.
The instantiation phase of this recovery action should be carried out in accordance with the redundancy model configuration of the various service groups that belong to the application.
This action is performed as a consequence of an SA_AMF_APPLICATION_RESTART recommended recovery action, which should be specified when the failure is deemed to be a global application failure.

3.11.1.3.4 Cluster Reset Recovery Action

The cluster should be reset. In order to execute this function, the Availability Management Framework reboots all nodes that are part of the cluster by using a low level interface without trying to terminate the components individually. To be effective, this operation must be performed such that all AMF nodes are first terminated before any of the AMF nodes starts to instantiate again. This recommendation should be used only in the rare case in which a component (most likely itself involved in error management) has enough knowledge to foresee a "cluster reset" as the only viable recovery action from a global failure. This action is performed as a consequence of an SA_AMF_CLUSTER_RESET recommended recovery action.

3.11.1.4 Repair

Repair is the action performed on erroneous entities (that is, entities with a disabled operational state) to bring them back into a healthy state (that is, to the enabled operational state).

One Availability Management Framework configuration attribute at node level (saAmfNodeAutoRepair of the SaAmfNode object class, see Section 8.7) and another one at service group level (saAmfSGAutoRepair of the SaAmfSG object class, see Section 8.9) specify whether the Availability Management Framework engages in **automatic repair** or not. The saAmfSGAutoRepair attribute applies to any service unit of the particular service group, and the saAmfNodeAutoRepair attribute applies only to the particular node.

If saAmfSGAutoRepair or saAmfNodeAutoRepair is turned on, the Availability Management Framework performs an automatic repair action after undertaking some recovery actions at the service unit or node level.
If an automatic repair configuration attribute is turned off (saAmfNodeAutoRepair or saAmfSGAutoRepair set to SA_FALSE), the Availability Management Framework performs no automatic repair action at the corresponding level, and it is the responsibility of applications with repair capabilities or of system administrators to perform repair actions (which are not under the Availability Management Framework's control) and then reenable the appropriate operational states when the repair is successfully completed.

Reenabling disabled entities can be performed in one of two ways:

- at the service unit or node level by executing the SA_AMF_ADMIN_REPAIRED administrative operation;
- at the component level by invoking the saAmfComponentErrorClear_4() function.

It is expected that repair actions bring the repaired entities (components or service units) in the uninstantiated presence state before reenabling the appropriate operational states.

Note that combined recovery and repair actions like the node failfast are also disabled when saAmfSGAutoRepair is set to SA_FALSE.

The Availability Management Framework treats the component and service unit restart recovery actions, which are described in Section 3.11.1.3.1, as repair actions and does not require any additional repair action in this case. The Availability Management Framework reenables the operational state of the component or of the service unit when the restart operation completes successfully.

In the case of a component fail-over recovery action and regardless of any configuration attribute setting, the Availability Management Framework always tries to reinstantiate the erroneous component; if it is successful, it reenables the erroneous component. The Availability Management Framework performs these actions to avoid leaving a service unit partially disabled for an indefinite amount of time. If the instantiation of the erroneous component fails, the Availability Management Framework sets the operational state of the service unit to disabled.

If a node leaves the cluster membership while the Availability Management Framework is performing an automatic repair action on a service unit of that node, the fact that the node leaves the cluster membership supersedes the service unit repair action, and the Availability Management Framework considers the repair action completed when the node rejoins the cluster membership.
However, if a node leaves the cluster membership while the Availability Management Framework is performing an automatic repair action on that node, the fact that the node leaves the cluster membership may not eliminate the need for the node repair action, and the Availability Management Framework may need to complete the repair action when the node rejoins the cluster membership, if the node has not been rebooted in the meantime.

3.11.1.4.1 Recovery and Associated Repair Policies

In this section, the recovery policies and the associated automatic repair policies are presented.

- **Service Unit Fail-Over Recovery**—In the context of a service unit fail-over recovery action, the Availability Management Framework attempts to terminate all components of the service unit. If the service group containing the service unit has the automatic repair configuration attribute set (`saAmfSGAutoRepair` set to SA_TRUE), and all components have been successfully terminated, the Availability Management Framework reenables the operational states of the service unit and of its disabled components and evaluates the various criteria used to determine if the service unit must be reinstantiated (such as the preferred number of in-service service units for the service group containing that service unit); it then reinstantiates service units, if deemed necessary.
- **Node Switch-Over**, **Node Fail-Over** and **Node Failfast Recovery**—After a node switch-over or node fail-over recovery action, if the erroneous node has the automatic repair configuration attribute set (`saAmfNodeAutoRepair` set to SA_TRUE), the Availability Management Framework reboots the node. The Availability Management Framework treats a node failfast recovery action as a repair action and does not require any additional repair action in this case. When such a node rejoins the cluster, the Availability Management Framework reenables its operational state and the operational state of its disabled service units and components (except for components with the termination-failed presence state). It then evaluates the various criteria used to determine if service units of that node must be reinstantiated (such as the preferred number of in-service service units service groups that have service units on that node) and reinstantiates service units if deemed necessary.

The following table describes the recovery policies and the associated automatic repair policies.

Table 17 Recovery and Associated Automatic Repair Policies

Recovery Action	Automatic Repair
service unit fail-over	Availability Management Framework attempts to instantiate the service unit
node switch-over	node reboot
node fail-over	node reboot
node failfast	none—already part of recovery

3.11.1.4.2 Restrictions to Auto-Repair

It is imperative that under certain circumstances the Availability Management Framework must not engage auto-repair actions. One such instance is during a software upgrade campaign, as defined by the Software Management Framework specification ([8]). In this case, the Availability Management Framework must be explicitly prevented from undertaking an automatic repair action to enable the initiator of the upgrade campaign to take some corrective or alternate actions like suspending the campaign in case components fail when they are being upgraded.

In order to disable the auto-repair behavior of the Availability Management Framework on a selective basis for components and containing service units, the service unit configuration in the Availability Management Framework Information Model supports a configuration attribute called `saAmfSUMaintenanceCampaign` (see the `SaAmfSU` object class in Section 8.10 on page 350), which can be modified to instruct the Availability Management Framework about disengaging auto-repair under various circumstances.

Note that the Availability Management Framework treats the component and service unit restart actions, which are described in Section 3.11.1.3.1, as well as node failfast recovery actions as repair actions that can also be disabled by setting the `saAmfSUMaintenanceCampaign` configuration attribute.

The `saAmfSUMaintenanceCampaign` configuration attribute contains the name of the maintenance campaign that is being currently run. When this attribute holds a valid value for a particular service unit, the Availability Management Framework disables the service unit without attempting any sort of repair in case constituent components fail. Additionally, all operational state change notifications (see Section 11.2.2.2 on page 429) pertinent to that service unit contain an indication that the service unit is involved in a maintenance (or upgrade) campaign.

3.11.1.5 Recovery Escalation

When an error is reported on a component, the error report also contains a recommended recovery action. The Availability Management Framework decides whether the recommended recovery action is executed, rejected, or escalated. The **recovery escalation** covers cases in which the recovery action is too weak to prevent further errors. The underlying principle of the escalation is to progressively extend the scope of the error from component to service unit, and from service unit to node (that is, considering more and more entities to be involved in the error that shows up in a component).

3.11.2 Recovery Escalation Policy of the Availability Management Framework

3.11.2.1 Recommended Recovery Action

The following **recommended recovery actions** are defined in the `SaAmfRecommendedRecoveryT` enum (see Section 7.4.7 on page 257) and can be specified in the `saAmfComponentErrorReport_4()` API (refer to Section 7.12.1 on page 325):

- `SA_AMF_NO_RECOMMENDATION`: used when the scope of the error is unknown. In this case, the Availability Management Framework engages the configured recovery policy for the component. This recovery policy is specified by the `saAmfCompRecoveryOnError` configuration attribute, defined in the `SaAmfComp` object class (see Section 8.13.2).
- `SA_AMF_COMPONENT_RESTART`: used when the scope of the error is the component.
- `SA_AMF_CONTAINER_RESTART`: used when the scope of the error is a container component and all collocated contained components. This recommended recovery action can only be requested for a contained component in order to restart the associated container.
- `SA_AMF_COMPONENT_FAILOVER`: used when the error is related to the execution environment of the component on the current node.
- `SA_AMF_NODE_SWITCHOVER`, `SA_AMF_NODE_FAILOVER`, and `SA_AMF_NODE_FAILFAST`:
  these three recommended recovery actions are used when the error has been identified as being at the node level, and components should not be in service on the node. They indicate different levels or urgency to move the service instances out of the node.
- `SA_AMF_APPLICATION_RESTART`: used when the error has been identified as a global application failure.

- SA_AMF_CLUSTER_RESET: used when the error has been identified at the cluster level.

The Availability Management Framework validates the recommended recovery action in an implementation-dependent way. This could be done, for example, by putting in place security measures like access control and authentication schemes. If the validation succeeds, the Availability Management Framework will not implement a weaker recovery action than the recommended one; however, the Availability Management Framework may decide to implement a stronger recovery action based on its recovery escalation policy. If the validation fails, the Availability Management Framework rejects the error report with the return code SA_AIS_ERR_ACCESS unless the recommended recovery action is SA_AMF_NO_RECOMMENDATION.

The following levels of escalation are implemented by the Availability Management Framework:

Table 18 Levels of Escalation

Escalation Level	Recommendation	Escalated to
1	SA_AMF_COMPONENT_RESTART	service unit restart (see Section 3.11.1.3.1)
2	SA_AMF_COMPONENT_RESTART	SA_AMF_COMPONENT_FAILOVER
3	SA_AMF_COMPONENT_RESTART or SA_AMF_COMPONENT_FAILOVER	SA_AMF_NODE_FAILOVER

3.11.2.2 Escalations of Levels 1 and 2

If some components of the same service unit fail and are restarted too many times within a given time period (the probation period), the Availability Management Framework escalates the recovery to a restart of the entire service unit. If, after this first level of escalation, the service unit is restarted too many times in a given time period because of failures of its components, the Availability Management Framework performs the SA_AMF_COMPONENT_FAILOVER recovery action, which is described as “Component or Service Unit Fail-Over” on page 195.

The remainder of this section provides a detailed explanation on how AMF implements the first two escalation levels.

Each service group can be configured with the following attributes, which are defined in the SaAmfSG object class (see Section 8.9):

- saAmfSGCompRestartProb (time value)
- saAmfSGCompRestartMax (maximum count)
- saAmfSGSuRestartProb (time value)
- saAmfSGSuRestartMax (maximum count)

The escalation policy algorithm for **escalations of levels 1 and 2** starts when an error with an SA_AMF_COMPONENT_RESTART recommended recovery action is received by the Availability Management Framework for a component of a particular service unit, and the service unit is not already in the middle of a probation period (neither "component restart" nor "service unit restart" probation period, see below).

At this time, the Availability Management Framework considers that it is at the beginning of a new "component restart" probation period for that service unit. The Availability Management Framework starts counting the number of components of that service unit it has to restart due to an error report with an SA_AMF_COMPONENT_RESTART recommended recovery action.
Components restarted due to dependencies (see Section 3.8.2) should not be counted.

If this count does not reach the saAmfSGCompRestartMax value before the end of the "component restart" probation period (the duration of the period is specified by saAmfSGCompRestartProb), the "component restart" probation period for the affected service unit expires.
It will be reinitiated when the Availability Management Framework receives the next occurrence of an error with an SA_AMF_COMPONENT_RESTART recommended recovery action for a component of the particular service unit.
If this count reaches the saAmfSGCompRestartMax value before the end of the "component restart" probation period, the Availability Management Framework performs the first level of recovery escalation for that service unit: the Availability Management Framework restarts the entire service unit.

At this time, the Availability Management Framework considers that escalation of level 1 is active for this service unit and terminates the current "component restart" probation period for the service unit. At the same time, it starts the "service unit restart" probation period for the service unit.
During the "service unit restart" probation period, each error report on the service unit with an SA_AMF_COMPONENT_RESTART recommended recovery action immediately escalates the recovery to an entire service unit restart (as level 1 escalation is active).
When the "service unit restart" probation period starts, the Availability Management

Framework also starts counting the number of times it has to perform a level 1 escalation.
If this count does not reach the `saAmfSGSuRestartMax` value before the end of the "service unit restart" probation period (the duration of the period is specified by `saAmfSGSuRestartProb`), the "service unit restart" probation period for the affected service unit expires.
If this count reaches the `saAmfSGSuRestartMax` value before the end of the "service unit restart" probation period, the Availability Management Framework performs the second level of recovery escalation for that service unit: the Availability Management Framework fails over the entire service unit and terminates the "service unit restart" probation period.

Container and contained components will follow the same recovery actions as described above with the following difference: when a container component is restarted, this recovery action triggers the restart of its associated contained components (see Section 3.11.1.3.1).

The count of restarted components of the service unit during the `saAmfSGCompRestartProb` probation period is not increased when contained components are restarted as a consequence of the restart of the associated container component. Similarly, the count of restarts of the service unit during the `saAmfSGSuRestartProb` probation period is not increased when the service unit containing the contained components is restarted as a consequence of the restart of the associated container component.

Note: The first-level escalation of the `SA_AMF_CONTAINER_RESTART` recommended recovery action requested for a contained component is the restart of the service unit containing the associated container component, which triggers the restart of the service units containing contained components associated with the container component.
The second-level escalation of the `SA_AMF_CONTAINER_RESTART` recommended recovery action requested for a contained component is the fail-over of the service unit containing the associated container component, which triggers the fail-over of the service units containing contained components associated with the container component.

Regarding the order of recovery operations of service instances protected by service groups containing container components and the order of recovery operations of service instances protected by service groups containing associated contained components when failing over a container component, refer to the recommendation in Section 6.3.

3.11.2.3 Escalation of Level 3

If the Availability Management Framework fails over too many service units out of the same node in a given time period as a consequence of error reports with either SA_AMF_COMPONENT_RESTART or SA_AMF_COMPONENT_FAILOVER recommended recovery actions, the Availability Management Framework escalates the recovery to an entire node fail-over.

The Availability Management Framework maintains the following configuration parameters on a per-node basis, which are used to implement **escalations of level 3**.

- saAmfNodeSuFailOverProb
- saAmfNodeSuFailoverMax

These attributes are defined in the SaAmfNode object class (see Section 8.7).

The escalation algorithm of level 3 is very similar to the algorithm applied for levels 1 and 2.

The escalation policy algorithm for an escalation of level 3 starts when the Availability Management Framework performs a service unit fail-over as a consequence of an escalation of level 2 or of an error report with an SA_AMF_COMPONENT_FAILOVER recommended recovery action on a node that is not already in the middle of a “service unit fail-over” probation period.

At this time, the Availability Management Framework considers that it is at the beginning of a new “service unit fail-over” probation period for that node. The Availability Management Framework starts counting the number of service unit fail-overs it has to perform on that node as a consequence of an escalation of level 2 or of an error report with an SA_AMF_COMPONENT_FAILOVER recommended recovery action.

If this count does not reach the saAmfNodeSuFailoverMax value before the end of the “service unit fail-over” probation period (the duration of the period is specified by saAmfNodeSuFailOverProb), the “service unit fail-over” probation period is terminated for all service units of the affected node.

If this count reaches the saAmfNodeSuFailoverMax value before the end of the “service unit fail-over” probation period, the Availability Management Framework performs the third level of recovery escalation for the node: the Availability Management Framework fails over the entire node.

4 Local Component Life Cycle Management Interfaces

The SA Forum has adopted a model for **component life cycle** similar to what is currently done in other clustering products. The SA Forum defines a set of command line interfaces (CLI), which are provided by local components to enable the Availability Management Framework to control their life cycles. In the remainder of this document, this interface will be referred to as the **Component Life Cycle Command Line Interface** (CLC-CLI).

Five CLC-CLI commands are included in this specification: `INSTANTIATE`, `TERMINATE`, `CLEANUP`, `AM_START`, and `AM_STOP`.

4.1 Common Characteristics

CLC-CLI commands and associated configuration attributes are part of the component configuration, as defined for the Availability Management Framework.

The CLC-CLI configuration contains the following attributes:

- the pathname of the CLC-CLI command (see Section 4.2),
- the list of environment variables (see Section 4.3) and arguments (see Section 4.4) to be provided to the CLC-CLI command by the Availability Management Framework at runtime, and
- a timeout value used to control the execution of the CLC-CLI command (refer to the sections describing each CLC-CLI command). The Availability Management Framework considers that the CLC-CLI command failed if it did not complete in the time interval specified by this timeout.

Additional information on CLC-CLI configuration attributes is provided in Chapter 8 and [7].

CLC-CLI commands are idempotents.

4.2 Configuring the Pathname of CLC-CLI Commands

The **pathname of a CLC-CLI command** of a component is obtained by specifying the **pathname prefix** of a location, which applies to all CLC-CLI commands of the component, and a **per-command pathname** to each of the CLC-CLI commands, which is relative to this pathname prefix.

The pathname prefix is both AMF node-specific and software bundle-specific, as it depends on the installation location of the software bundle on a particular AMF node. For the definition of a software bundle, see [8]. Each AMF node has an association with the software bundles installed on it. This association is described by the `SaAmfNodeSwBundle` object class (shown in Section 8.7). The `saAmfNodeSwBundlePathPrefix` attribute of an `SaAmfNodeSwBundle` object defines the root installation directory for a particular software bundle on a given AMF node. This pathname prefix applies to all components of component types that refer to this software bundle when these components are mapped to this AMF node (for more details on the mapping, see Section 3.1.9). A component's type is given in the `saAmfCompType` attribute (shown in Section 8.13.2), which is the DN of an object of the `SaAmfCompType` object class (shown in Section 8.13.1). The `saAmfCtSwBundle` attribute of this component type object contains the name of the software bundle that is required for the component on the AMF node. The pathname prefix for this software bundle on the selected AMF node needs to be used with the CLC-CLI commands of the component.

The attributes referring to the per-command relative pathname are named `saAmfCtRelPath<commandName>Cmd`; they are defined in the `SaAmfCompType` object class (shown in Section 8.13.1). The actual names of these attributes are obtained by replacing each `<commandName>` with a name representing the specific CLC-CLI command (`AMStart`, `AMStop`, `Cleanup`, `Instantiate`, and `Terminate`).

The absolute pathname for a CLC-CLI command for a component is the concatenation of the pathname prefix of the software bundle on the given AMF node (`saAmfNodeSwBundlePathPrefix`) and the appropriate per-command relative pathname of the component type (`saAmfCtRelPath<commandName>Cmd`).

When the Availability Management Framework decides to execute a CLC-CLI command for a local component on a particular AMF node, it performs the following steps:

1. Using the `saAmfCompType` attribute of the component, it looks up the component's type described by an `SaAmfCompType` object.
   The component type object specifies
   - the required software bundle in the `saAmfCtSwBundle` attribute and
   - the relative paths to the CLI-CLI commands through the `SaAmfCtRelPath<commandName>Cmd` attributes.

2. For the selected AMF node, it looks up the pathname prefix for the software bundle (`saAmfCtSwBundle`), as given in the `saAmfNodeSwBundlePathPrefix` attribute of the appropriate `SaAmfNodeSwBundle` association object.
3. It concatenates this pathname prefix of the software bundle (`saAmfNodeSwBundlePathPrefix`) with the per-command relative pathname of the component type (`saAmfCtRelPath<commandName>Cmd`) to compose the absolute pathname for the CLC-CLI command.

4.3 CLC-CLI Environment Variables

An SA-aware component obtains the name value pairs for each component service instance assigned to itself or to the components for which it is 'proxying' when the `saAmfCSISetCallback()` callback function is invoked (see Section 7.9.2). In case of a non-proxied, non-SA-aware component, the Availability Management Framework passes the name/value pairs of the component service instance as environment variables to each CLC-CLI command.

The `SA_AMF_COMPONENT_NAME` environment variable is set in the environment of each CLC-CLI command. This environment variable contains the name of the component the CLC-CLI command is acting upon.

To avoid non printable values for environment variables, values containing unicode characters (such as component names) are encoded by the Availability Management Framework in the following way:

- first, the unicode characters are translated into UTF-8 encoding, as described in RFC 2253 ([11]), to obtain a character string;
- then, the quoted-printable encoding from RFC 2045 ([12]) is used to substitute non-printable characters in the string.

The `saAmfCompCmdEnv` configuration attribute of the `SaAmfComp` object class (shown in Section 8.13.2) defines environment variables and their values for all CLC-CLI commands of the component. These environment variables are added to the environment variables specified for components of this type (see the `saAmfCtDefCmdEnv` configuration attribute defined in the `SaAmfCompType` object class, shown in Section 8.13.1).
If the `saAmfCompCmdEnv` attribute is not specified, only the environment variables (if any) and their values in the `saAmfCtDefCmdEnv` attribute apply.

4.4 Configuring CLC-CLI Arguments

The attributes symbolically named `saAmfComp<commandName>CmdArgv` in the `SaAmfComp` object class, shown in Section 8.13.2, and the attributes symbolically named `saAmfCompCtDef<commandName>CmdArgv` in the `SaAmfCompType` object class, shown in Section 8.13.1, are used to configure arguments of CLC-CLI commands. The actual names of these attributes are obtained by replacing each `<commandName>` with a name representing the specific CLC-CLI command (`AMStart`, `AMStop`, `Cleanup`, `Instantiate`, and `Terminate`). The configuration attributes `saAmfComp<commandName>CmdArgv` of a component denote arguments of the corresponding CLC-CLI commands of the component. These arguments are added to the arguments specified in the corresponding `saAmfCompCtDef<commandName>CmdArgv` attributes of the component type to which the component pertains. If an `saAmfComp<commandName>CmdArgv` attribute is not specified, only the arguments (if any) specified in the corresponding `saAmfCompCtDef<commandName>CmdArgv` attribute are used.

4.5 Exit Status

The valid range for the **exit status** is

0 <= exit status <= 255.

CLC-CLI commands have a zero exit status in case of success and nonzero in case of failure. Values in the range

200 <= exit status <= 254

have either predefined meanings or are reserved for future usage.

The reaction of the Availability Management Framework to these errors is described for each CLC-CLI command in the next sections.

4.6 INSTANTIATE Command

The Availability Management Framework must run the `INSTANTIATE` command when it decides to instantiate a new instance of a local component, except for proxied and contained components. This command must not be used for proxied or contained components: a proxied component must be instantiated by its proxy component, and a contained component must be instantiated by its associated container component.

The `INSTANTIATE` command may create none or several processes, files, shared memory segments, and so on.

Note that some components may not have any processes and the `INSTANTIATE` command may be limited to some administrative action such as configuring an IP address on the local node or mounting a file system.

The `INSTANTIATE` command must report success if the component is already instantiated when the command is run. If the `INSTANTIATE` command is completed successfully, the component must be fully instantiated. The timeout associated with the `INSTANTIATE` command is used to set a limit on the time the Availability Management Framework will give for the component instantiation to complete. This time includes the completion of the `INSTANTIATE` command itself; for SA-aware components, it also includes the extra time that may be needed by the component, after `INSTANTIATE` returns, to register with the Availability Management Framework. Hence, for SA-aware components, this timeout sets a time limit for the newly instantiated component to register.

`INSTANTIATE` must return a nonzero exit status if the component is not instantiated successfully. If `INSTANTIATE` returns a nonzero exit status (even if it is outside the range valid for the Availability Management Framework, as described in Section 4.5), or the instantiation of the component does not complete in the time period specified by the `INSTANTIATE` timeout, the Availability Management Framework generates an error report on the failed component and runs the `CLEANUP` command described in Section 4.8 to perform all necessary cleanup.

The Availability Management Framework makes few attempts to recover from this error by trying to reinstantiate the component if restart is not disabled. The Availability Management Framework first makes a configurable number of attempts to immediately reinstantiate the component, followed by a configurable number of attempts to reinstantiate the component with a configurable delay between each attempt. The following configuration attributes of the `SaAmfComp` object class (shown in Section 8.13.2) are used to configure these number of attempts and the delay between each attempt: `saAmfCompNumMaxInstantiateWithoutDelay`, `saAmfCompNumMaxInstantiateWithDelay`, and

saAmfCompDelayBetweenInstantiateAttempts, respectively.
An attempt to reinstantiate the component fails if the INSTANTIATE command returns an error or fails to complete in the configured timeout period (specified by the saAmfCompInstantiateTimeout configuration attribute of the SaAmfComp object class, shown in Section 8.13.2).
If all these attempts fail, the Availability Management Framework has the possibility to force a node failfast recovery action. This possibility is controlled by the saAmfNodeFailfastOnInstantiationFailure configuration attribute of the SaAmfNode object class, shown in Section 8.7. The node failfast includes an implicit node reboot, which transitions the presence state of all local components of the node (including its hardware components) to uninstantiated. For more details, see Section 3.11.1.3.

If node reboot is disabled, or if a single reboot did not solve the problem, the Availability Management Framework sets the operational state of the component to disabled and its presence state to instantiation-failed. The presence state of the enclosing service unit becomes also instantiation-failed (it may also become termination-failed if other components of the service units failed to terminate successfully; note that the termination-failed state overrides the instantiation-failed state in this case). The Availability Management Framework performs a service unit level recovery action if the error occurred when some service instances were already assigned or being assigned to the service unit; however, no further automatic repair for this service unit beyond the already attempted node reboot is provided, and an explicit action from an entity external to the Availability Management Framework is required to repair the service unit (for more details, see Section 3.11.1.4 on page 197).

The following error code is recognized by the Availability Management Framework:

SAF_CLC_NO_RETRY (200): the error that occurred when attempting to instantiate this component is persistent, and no retries or node reboot should be attempted.

4.7 TERMINATE Command

The Availability Management Framework terminates an SA-aware or a proxied, pre-instantiable component by invoking the `saAmfComponentTerminateCallback()` callback function on the component itself or on its proxy component, respectively. The Availability Management Framework terminates a proxied, non-pre-instantiable component, by invoking the `saAmfCSIRemoveCallback()` function of its proxy component for the CSI of the proxied component.

However, when the Availability Management Framework needs to stop a service provided by a non-proxied, non-SA-aware component, or needs to terminate such a component, no callback can be invoked, and the Availability Management Framework executes the `TERMINATE` command. The `TERMINATE` command should stop the service being provided in such a way that the service can be resumed by another instance of the same component or another component with minimal disruption.

This CLC-CLI command is mandatory for all local non-proxied, non-SA-aware components and must not be used for SA-aware components and proxied components.

When the `TERMINATE` command completes successfully, it must leave the component uninstantiated. The uninstantiated state of a local component can be defined as the state of the component just after a node reboot and before the `INSTANTIATE` command is run. `TERMINATE` should succeed if the component is not instantiated when the command is run.

The `TERMINATE` command should release all resources allocated by the component.

The `TERMINATE` command must return an error if the component is not fully terminated, or if some resources could not be released. If the `TERMINATE` command returns an error or fails to complete in the configured timeout period (specified by the `saAmfCompTerminateTimeout` configuration attribute of the `SaAmfComp` object class, shown in Section 8.13.2), the Availability Management Framework runs the `CLEANUP` command to perform all necessary cleanup actions.

4.8 CLEANUP Command

When recovering from errors, the Availability Management Framework does not trust erroneous components to execute any callbacks, but it still needs a method to terminate the particular instance of a component with the minimum interaction with the component itself. Such a method is also needed when either the callbacks or the TERMINATE command (see Section 4.7) used to terminate the component fail. In this case, the Availability Management Framework forces the cleanup of the component by invoking the pertinent callback or by running the CLEANUP command (see Table 37 in Appendix A).

The CLEANUP command is mandatory for all local components (proxied or non-proxied), except for contained components, and it must not be used for external or contained components.

When the CLEANUP command completes successfully, it must leave the component uninstantiated. CLEANUP should succeed if the component is not instantiated when the command is run.

CLEANUP should perform any cleanup of resources allocated by the component and should execute under the assumption that the component may be in an erroneous state in which it cannot actively perform any cleanup actions itself. CLEANUP must return an error if the component is not fully terminated, or if some necessary cleanup could not be performed. If the component has been configured with a monitor (see AM_START in Section 4.9), the CLEANUP command also needs to clean up any resources that the AM_STOP command may have failed to clean up.

If the CLEANUP command returns an error or fails to complete in the configured timeout period (specified by the saAmfCompCleanupTimeout configuration attribute of the SaAmfComp object class, shown in Section 8.13.2), the Availability Management Framework has the possibility to force a node failfast recovery action. This possibility is controlled by the saAmfNodeFailfastOnTerminationFailure configuration attribute of the SaAmfNode object class, shown in Section 8.7. The node failfast includes an implicit node reboot, which places all local components of the node (including its hardware components) into the uninstantiated presence state. For more details, see Section 3.11.1.3.

If the node reboot is not allowed by the configuration of the node, the Availability Management Framework sets the component's operational state to disabled and the component's presence state to termination-failed. The presence state of the enclosing service unit becomes also termination-failed, and its operational state becomes disabled. No further automatic repair is attempted by the Availability Management Framework for that service unit, and an explicit action from an entity external to the

Availability Management Framework is required to repair the service unit (for more details, see Section 3.11.1.4 on page 197).

If the component and any of its contained components (for a container component) were assigned the active HA state for some component service instances when the CLEANUP command was executed, and semantics of the redundancy model of its enclosing service group guarantee that at a point in time only one component can be in the active HA state for a given component service instance, the failure to terminate that component prevents the Availability Management Framework from assigning to another component the active HA state for these component service instances (and by the same token prevents the assignment of the active HA state to other service units for the service instances that contain the involved CSIs). In this case, the service instances will stay unassigned until an administrative action is performed to terminate the failed component.

4.9 AM_START Command

The Availability Management Framework executes the AM_START command after the component has been successfully instantiated or to resume monitoring after the command has been stopped by some administrative operations. The monitor processes started by AM_START should periodically assess the health of the component and report any error by invoking the saAmfComponentErrorReport_4() function.

The AM_START command is optional for all local components and must not be used for external components.

If the AM_START command returns an error or fails to complete in the configured timeout period (specified by the saAmfCompAmStartTimeout configuration attribute of the SaAmfComp object class, shown in Section 8.13.2), the Availability Management Framework will retry a few times to start the monitor. If AM_START did not complete in the configured timeout period, the Availability Management Framework runs AM_STOP before running AM_START again. If after a configurable amount of retries (the maximum number of retries is specified by the saAmfCompNumMaxAmStartAttempts configuration attribute of the SaAmfComp object class), the Availability Management Framework fails to start the monitor, the Availability Management Framework reports an error on the component level.

4.10 AM_STOP Command

The Availability Management Framework runs the AM_STOP command when active monitoring of the component must be stopped. The Availability Management Framework stops active monitoring before terminating a component and when directed to do so by administrative operations.

The AM_STOP command is mandatory for components that have an AM_START command, and must not be used for components that do not have an AM_START command.

If the AM_STOP command returns an error or fails to complete in the configured timeout period (specified by the saAmfCompAmStopTimeout configuration attribute of the SaAmfComp object class, shown in Section 8.13.2), the Availability Management Framework will retry a few times to stop the monitor. If AM_STOP is invoked in the context of a component termination, and if AM_STOP still fails after all retries (the maximum number of retries is given by the saAmfCompNumMaxAmStopAttempts configuration attribute of the SaAmfComp object class), the Availability Management Framework terminates the component and cleans it up to ensure that the monitor eventually gets stopped. If AM_STOP fails while the Availability Management Framework tries to terminate a component in the context of a recovery action, the Availability Management Framework may skip the retries and go ahead immediately by terminating the component.

4.11 Usage of CLC-CLI Commands Based on the Component Category

Table 19 Usage of CLC-CLI Commands Based on the Component Category

CLC-CLI Command	Mandatory	Forbidden	Optional
INSTANTIATE	non-proxied, local components, except for contained components	contained and proxied components	-
TERMINATE	non-proxied, non-SA-aware components	SA-aware and proxied components	-
CLEANUP	local components, except for contained components	contained and external components	-
AM_START AM_STOP	-	external components	local components

For further details on component categories, refer to Table 3 on page 50.

5 Proxied Components Management

This chapter summarizes information presented in other chapters on the management of proxy and proxied components and also provides additional information on the management of these components.

For a comparison of the management of proxied components with the management of contained components, refer to Section 6.4.

5.1 Properties of Proxy and Proxied Components

The following list summarizes the main properties of proxy and proxied components.

- A single proxy component can mediate between the Availability Management Framework and multiple proxied components.
- Although the proxied/proxy approach is recommended when the proxied components are not located on AMF nodes, it can also be applied when the proxied components are contained in local service units.
- Pre-instantiable proxied components cannot be located in the same service unit as their proxy components. This assumption is devised to prevent potential cyclic dependencies when service units are instantiated.
- If the proxy and proxied local components are hosted in different service units, these service units may reside on different AMF nodes.
- The configuration of proxy/proxied components must include information about the association of a proxied component to the CSI through which the proxied component will be proxied (termed proxy CSI). In other words, a proxied component configuration has a configuration attribute (`saAmfCompProxyCsi` in the `SaAmfComp` object class, shown in Section 8.13.2) which contains the name of the CSI through which the proxied component will be proxied.
- A proxy CSI can be dedicated to “proxy” one or more proxied components.
- A proxy component can be configured to accept multiple CSIs; one or more for “proxying” proxied components and others for providing non-proxy services. Note that in terms of function there is no difference between a proxy CSIs and other CSIs. The proxy CSI corresponds to the workload of “proxying” a proxied component.
- Only the proxy component with the active HA assignment for a proxy CSI may register the proxied components associated with the CSI.
- The redundancy model (for a discussion of this notion, refer to Section 3.6) of the proxy component can be different from redundancy models of its proxied components.

5.2 Life Cycle Management of Proxied Components

By means of the proxy CSI configuration (see the preceding section), the Availability Management Framework determines the associations amongst proxy and proxied components.

After the Availability Management Framework successfully assigns a proxy CSI with active HA state to a proxy component, the Availability Management Framework requests this proxy component to instantiate the corresponding pre-instantiable proxied components by invoking the `SaAmfProxiedComponentInstantiateCallbackT` function.
This step is not applicable to non-pre-instantiable components, as the instantiation of a non-pre-instantiable proxied component will be done by its proxy component when the Availability Management Framework assigns the CSI with active HA state to the proxied component.

After a proxied component is instantiated, the respective proxy component must register the proxied component with the Availability Management Framework (see also Section 7.1.1). Just like an SA-aware component, a proxied component is considered to be fully instantiated only after the registration of the proxied component is successful. After registration of a pre-instantiable proxied component, the Availability Management Framework can assign one or more CSIs to it (through the respective proxy component) with appropriate HA states. If the instantiation of a proxied component fails, that is, the proxy component returns an error to the instantiation request for a pre-instantiable proxied component or to the assignment of a CSI with the HA active state to a non-pre-instantiable proxied component, the Availability Management Framework must attempt to revive the failed proxied component by cleaning it up and reinstantiating it as it does for an SA-aware component. Refer also to Appendix C, which uses a sample configuration to illustrate a typical proxy and pre-instantiable proxied instantiation and registration sequence, as explained in this section.

When a component registers another component, the Availability Management Framework shall verify if the component invoking the registration is a proxy and has the active assignment for the CSI through which the component being registered can be proxied. If not, the Availability Management Framework will assume that the calling component does not have the authority to register the proxied component and will return the error code `SA_AIS_ERR_BAD_OPERATION` (for the component registration interface, refer to Section 7.6.1).

5.3 Proxy Component Failure Handling

If a proxy component fails, the Availability Management Framework may perform a fail-over if fail-over is allowed by the redundancy model of the service group to which the proxy belongs. During the proxy component fail-over procedure, the Availability Management Framework implicitly unregisters all registered proxied components associated with the failing proxy component. However, this implicit unregistration should not be considered by the Availability Management Framework as a sign of a proxied component failure. The implicit unregistration simply indicates that the proxy component is unable to continue the “proxying” work, and the Availability Management Framework should find another proxy component to take over the “proxying” work.

If the Availability Management Framework can find another proxy component that is capable of “proxying” the given proxied components, it will make an active assignment of the proxy CSIs of the proxied components to this other proxy component. The selection of the proxy component to take over the proxy job is based on the redundancy model of the service group containing the proxy component. For example, for a proxy component contained in a service unit that pertains to a service group with the 2N redundancy model (termed here the proxy’s service group), the newly selected proxy component should be the one that had standby HA assignment for the proxy CSI (that is, the CSI through which the proxied components are proxied). A similar procedure is followed during a switch-over in the proxy’s service group.
In this case, if the newly selected proxy component succeeds in taking over the proxying task, it registers again the proxied component without an explicit instantiation step. If the proxy component is unable to take over the proxying task due to the failure of the proxied component, it must report an error on the proxied component, so the Availability Management Framework can try to clean up the proxied component.

If the Availability Management Framework is unable to find another proxy component to “proxy” a given proxied component, the given proxied component shall enter the `SA_AMF_PROXY_STATUS_UNPROXIED` status (defined in Section 7.4.4.8), and an appropriate alarm shall be issued by the Availability Management Framework (see Section 11.2.1.5 on page 427) to indicate this situation. Whenever a proxied component enters the `SA_AMF_PROXY_STATUS_PROXIED` status (defined in Section 7.4.4.8), an appropriate notification (see Section 11.2.2.7 on page 434) will be issued to indicate the change in the status of the proxied component.

6 Contained Components Management

This chapter summarizes information presented in other chapters on the management of container and contained components and also provides additional information on the management of these components.

6.1 Overview of Container and Contained Components

This section serves as a guideline for the features of container and contained components.

6.1.1 Definitions

Refer to Section 3.1.2.1.1, which introduces the terms container component, contained component, associated container component, associated contained component, and collocated contained components. For the definition of the container CSI term, refer to Section 3.1.3 and Section 6.2.

6.1.2 Component Category

Container and contained components are local SA-aware components. See also Section 3.1.2.1.1.

6.1.3 Multiple Components per Process

Refer to Section 3.1.2.1.1.

6.1.4 Life Cycle Management of Contained Components

Refer to Section 6.2.

6.1.5 Container and Contained Components in Service Units and Service Groups

A service unit that contains a contained component can only contain collocated contained components (that is, they all have the same associated container component). For the association between these collocated contained components and the associated container component, a single container CSI is used (see Section 6.2).

Container components and contained components must not be located in the same service unit. The rationale for not mixing container and contained components in a service unit is that the container component must get its CSI assignments before CSIs are assigned to the contained components, because this is the mechanism used to determine the associated container component of a contained component, as explained in Section 3.1.3 and Section 6.2.

The rank of the service instance with the container CSI must not be lower than the ranks of the service instances having component service instances assigned to the associated contained components.

The rationale for not mixing contained components with components of other categories in a service unit is to be able to restart the associated container component as a recovery action to handle failures in associated contained components.

As any service unit of a service group can be assigned any service instance protected by the service group (see Section 3.1.6), and as it makes no sense to assign the container CSI configured for the contained components of a service unit to these contained components, service units containing contained components and service units containing container components must belong to different service groups. Section 6.1.6 discusses the redundancy models that are supported for these service groups.

The readiness state of a service unit containing contained components is affected by the HA state of the associated container component for their container CSI, as explained in Section 3.2.1.4.

AMF nodes and node groups configured for service units containing contained components (or for the corresponding service groups) must not conflict with the AMF nodes and with node groups configured for the service units containing container components that can potentially drive the life cycle of these contained components (or for the service groups containing these service units). This is because a container component and its associated contained components must reside on the same AMF node.
A service instance containing a container CSI should not contain any other component service instance.

6.1.6 Redundancy Models

For a discussion of redundancy models, refer to Section 3.6.

The redundancy model of service groups having service units containing a container component can be different from the redundancy model of service groups having service units containing the associated contained components.

The only redundancy model supported for service groups having service units containing a container component is the N-way active redundancy model. The reason for this decision is that only container components with the active HA assignment for a container CSI may handle the life cycle of contained components in cooperation with the Availability Management Framework (that is, may act as associated containers).

A service group containing contained components can be associated with any of the redundancy models defined by the Availability Management Framework.

6.1.7 Administrative Operations and Container and Contained Components

The description of the Availability Management Framework administrative operations is presented in Chapter 9. The peculiarities of these operations when container components and container CSIs are affected are described in Section 9.4.3 for the lock operation, in Section 9.4.6 for the shutdown operation, and in Section 9.4.7 for the restart operation.

Regarding the order of recovery operations for service instances protected by service groups containing container components and the order of recovery operations for service instances protected by service groups containing associated contained components, when certain administrative operations affect the container components, refer to the recommendation in Section 6.3.

6.1.8 Failure Handling

Refer to Section 6.3.

6.2 Life Cycle Management of Contained Components

The life cycle of contained components is handled by the Availability Management Framework in cooperation with the associated container component. The associated container component is responsible for instantiating and cleaning up associated contained components, and the Availability Management Framework is responsible for the termination of the contained components (see also Section 6.2.3).

6.2.1 Container CSI and Its Configuration

The Availability Management Framework supports the notion of **container CSI**. A container CSI represents the special workload of managing the life cycle of contained components.

A contained component must be configured with its container CSI, that is, the name of the container CSI must be specified in the `saAmfCompContainerCsi` configuration attribute of the `SaAmfComp` object class (shown in Section 8.13.2) of the contained component configuration.

The container CSI can contain information to be passed by the associated container component to the corresponding contained component. How this information is passed to the contained component is a private interface between container and contained components.

6.2.2 Assignment of the Container CSI

The Availability Management Framework determines—based on its configuration—to which container components the container CSI is assigned.

A single container component can handle the life cycle of one or more contained components in cooperation with the Availability Management Framework.

If there is just one container component on a node having the active HA state for a particular container CSI, this container component will be the associated container component of all contained components that are configured with this container CSI on this node, regardless of whether the contained components reside in one or multiple service units.

If the administrator wants to configure multiple service units containing contained components on a node, for example, *SU1*, *SU2*, and *SU3*, and these service units should be associated with the container components *ContainerC1*, *ContainerC2*, and *ContainerC3* (on the same node), respectively, the administrator can proceed as follows:

- Configure all components of *SU1* with the container CSI *ContainerCSI1*. Configure similarly all components of *SU2* with *ContainerCSI2* and all components of *SU3* with *ContainerCSI3*.
- Create three service groups *ContainerSG1*, *ContainerSG2*, and *ContainerSG3*, which have each one service unit on this node, and which are termed *ContainerSU1*, *ContainerSU2*, and *ContainerSU3,* respectively. The service units *ContainerSU1*, *ContainerSU2*, and *ContainerSU3* contain the container components *ContainerC1*, *ContainerC2*, and *ContainerC3,* respectively.
- Configure the following CSI assignments: *ContainerCSI1* to *ContainerC1*, *ContainerCSI2* to *ContainerC2*, and *ContainerCSI3* to *ContainerC3*.

If there are multiple container components on a node which have the active HA state for a particular container CSI, and one or more service units on the same node whose contained components are configured with the same container CSI, it is implementation-defined how the Availability Management Framework selects container components to handle the life cycle of the contained components of these service units. However, all contained components of a service unit must have the same associated container component.

Note that a container component can be configured to have multiple CSI assignments, one or more for handling contained components (container CSI) and others for providing other services. In terms of functionality and syntax, there is no difference between a container CSI used to determine the associated container component and CSI assignments corresponding to the workload of other services.

Actions taken by the Availability Management Framework when it changes or removes the HA state of a container component for a container CSI are described on page 81.

6.2.3 Life Cycle Callbacks

Two callback functions are invoked by the Availability Management Framework to control the life cycle of contained components.

- `SaAmfContainedComponentInstantiateCallbackT` (see Section 7.10.4) and
- `SaAmfContainedComponentCleanupCallbackT` (see Section 7.10.5).

The Availability Management Framework invokes the `SaAmfComponentTerminateCallbackT` callback function (see Section 7.10.1) directly on the associated contained component.

6.3 Failure Handling for Container and Contained Components

Changes of the operational state of a container component do not directly affect the operational state of its associated contained components. Note, however, that the change of the operational state of the container component can affect the associated contained components indirectly: if, for example, the operational state of a container component changes to disabled, the container component will be terminated, and this termination will affect the associated contained components, which will also be terminated.

Changes of the operational state of a contained component do not affect the operational state of either its associated container component or of its collocated contained components.

In case of a failure of a container component, the Availability Management Framework abruptly terminates the container component and assumes that the termination of the container component also forces the termination of all associated contained components. The Availability Management Framework does not individually terminate the associated contained components of a failed container component.

A failure of a container component is handled according to its own redundancy model, and the contained components are handled according to their own redundancy model. Note that the container and contained components can have different redundancy models.

If a contained component fails, it is the task of its associated container component to report an error on the failed component.

Recommendation: It is recommended that implementations of the Availability Management Framework recover the service instances that are protected by service groups containing contained components before attempting to recover the service instances protected by service groups containing the associated container components in case of fail-over, switch-over, and other scenarios that require a reassignment of service instances. This order is recommended to ensure minimal disruption of the service being provided by the contained components.

6.4 Proxied and Contained Components: Similarities and Differences

The main differences and similarities in how the Availability Management Framework handles proxy and proxied components on the one hand and container and contained component on the other hand are presented in the following list. To simplify the description, it is assumed that the proxy component is no container component.

- **Error Containment**: if a container component fails, the Availability Management Framework assumes that the termination of the container component also forces the termination of all associated contained components.
  The termination of a proxy component does not imply the termination of its proxied components.
- **SA-Aware components**: though the life cycle of a contained component is handled by the associated container in cooperation with the Availability Management Framework, a contained component is an SA-aware component, and it communicates directly with the Availability Management Framework.
  A proxied component is a non-SA-aware component, and it communicates with the Availability Management Framework only through its proxy component.
- **Recommended recovery:** contained components can specify the `SA_AMF_CONTAINER_RESTART` as a recommended recovery, which is a convenience method for contained components to recommend the component restart of their container. A counterpart for proxy components does not exist.
- **Limitations in the service unit configuration**: all contained components in a service unit need to be configured with the same container CSI. It is not permitted to configure contained components and non-contained components in the same service unit.
  A non-pre-instantiable proxied component and its proxy component can be located in the same service unit. A pre-instantiable proxied component and its proxy component must not be located in the same service unit.
- **Local Components**: both container and contained components are local components.
  A proxied component can be either a local or an external component.
- **Registration**: container and contained components register directly with the Availability Management Framework.
  Only the proxy component with the active HA assignment for a proxy CSI may register proxied components associated with the proxy CSI.

7 Availability Management Framework API

The Availability Management Framework API described in this chapter is based on the system description and the system model presented in Chapter 3. It provides the following services to application components.

- Library life cycle
- Component registration and unregistration
- Passive monitoring of processes of a component
- Component health monitoring
- Component service instance management
- Component life cycle
- Protection group management
- Error reporting
- Component response to Availability Management Framework requests

A component exists in a single service unit, and it typically consists of one or more processes executing on a node. It is the responsibility of the component to monitor and isolate faults within its scope and to generate error reports accordingly. As a function of these error reports, cluster membership changes, health monitor reports, and administrative operations, the Availability Management Framework manages internally the readiness state of the affected components. The Availability Management Framework drives the HA state of components on behalf of component service instances to provide service availability.

The function calls described in this chapter cover only the interactions between an SA-aware or a proxied component (through its proxy component) and the Availability Management Framework, and it does not cover operational or administrative aspects. Consequently, the logical entities that are represented in the parameters of the calls are limited to:

- SA-aware components
- Proxy components
- Proxied components (local or external)
- Component service instances
- Protection groups

The other logical entities, such as service units, service groups (including their redundancy model), and service instances are used when configuring the relationships among the components which must be maintained by the Availability Management Framework.

7.1 Availability Management Framework Model for the APIs

7.1.1 Callback Semantics and Component Registration and Unregistration

The Availability Management Framework issues requests to a component by invoking the callback functions provided by the component. A process of an SA-aware component intending to use the API functions of the Availability Management Framework must first initialize the Availability Management Framework library by invoking the `saAmfInitialize_4()` function, which is described in Section 7.5.1 on page 264. A handle is returned to the invoking process denoting this particular initialization of the Availability Management Framework library. One of the input parameters of the `saAmfInitialize_4()` function is the set of callback functions associated with this initialization.

One process of an SA-aware component registers with the Availability Management Framework for the component that it represents by invoking the `saAmfComponentRegister()` function (described in Section 7.6.1 on page 272) with the handle returned by the `saAmfInitialize_4()` function.

In addition, a contained component or a proxied component can be registered by one process of its container component or of its proxy component, respectively; the registration is done by invoking the `saAmfComponentRegister()` function and using either a newly initialized handle or the same handle used to register another component.
At most one process can register with the Availability Management Framework for a given component. This process is called the **registered process for the component**.

The registered process for a contained component may be the same registered process for its container component or a different process.
The registered process for a proxied component may be the same registered process for its proxy component or a different process.

Proxy and container components must be first registered before their proxied and contained components are registered.

There is no API function for a process to explicitly unregister a component. Unregistration of components is done automatically by the Availability Management Framework in the following situations:

⇒ When the presence state of the component becomes uninstantiated because the component has been terminated. This happens typically:

- after a successful execution of the `SaAmfComponentTerminateCallbackT` callback or the `CLEANUP` command for an SA-aware component or a proxied component, or
- when the active assignment of a component service instance is removed from a non-pre-instantiable proxied component.

⇒ When the Availability Management Framework successfully removes the active assignment of a proxy CSI from a proxy component. In this case, the Availability Management Framework unregisters all proxied components that had been registered by this proxy component while the proxy component was assigned the active HA state for the proxy CSI configured for these proxied components.

The Availability Management Framework also unregisters all components that are still registered with a particular handle when that handle is finalized explicitly by invoking the `saAmfFinalize()` function or implicitly when the process that initialized the handle exits. However, if an SA-aware component finalizes a handle that still has some registered components associated to it, the Availability Management Framework treats this finalization as an error of the SA-aware component. An SA-aware component should only finalize a handle when the previously associated registered components have automatically been unregistered by the Availability Management Framework, as indicated above.

A process that is part of a proxy component and that registers several proxied components may issue several calls to the `saAmfInitialize_4()` function to provide different sets of callback functions and obtain different handles that can be used to register the various proxied components.

Some of the callback functions are called by the Availability Management Framework only in the context of the registered process for a component. Additionally, there are other API functions that may be called only by the registered process for a component. The descriptions of the APIs of the Availability Management Framework in this chapter state these restrictions explicitly. Additionally, Appendix B provides a table showing which API functions can only be invoked by the registered process for a component and which API callback functions may only be invoked for such processes.

When the Availability Management Framework issues a request to a particular component, it triggers the invocation of a callback function. Some of the callback calls require a response from the component. In these cases, the component invokes the `saAmfResponse_4()` function (described in Section 7.13.1 on page 333) when it has successfully completed the action or has failed to perform the action.

More precisely, the following principles are applied in the Availability Management Framework/component interactions:

- The process is not required to complete the action requested by the Availability Management Framework within the invocation of the callback function. It may return from the callback function and complete the action later.
- The process is expected to notify the completion of the action (or any error that prevented it from performing the action) by invoking the `saAmfResponse_4()` function. The `saAmfResponse_4()` function must identify the callback action with which it is associated by providing the value of the `invocation` parameter that the Availability Management Framework supplied in the callback.
- Any function of the Availability Management Framework API, including `saAmfResponse_4()`, can be invoked from callback functions.

7.1.2 Component Healthcheck Monitoring

7.1.2.1 Overview

A component (or more specifically, each of its processes) is allowed to dynamically start and stop a specific **healthcheck**. Each healthcheck has an identification (**healthcheck key**) that is associated with a set of configuration attributes.

Healthchecks can be invoked by the Availability Management Framework or by the component.

The issue of potential transient overload caused by healthcheck invocations is not considered in this proposal. Overload is a global issue, and should be handled in a consistent global level; it will be considered in a future version of the AIS specification.

7.1.2.2 Variants of Healthchecks

There are two **variants of healthcheck,** depending on the invoker of the healthcheck:

- **Framework-invoked healthcheck**: for this variant of healthcheck, the Availability Management Framework invokes the `saAmfHealthcheckCallback()` callback periodically according to the healthcheck configuration attributes, described in Section 7.1.2.4. The Availability Management Framework expects the component to reply to an invoked healthcheck by calling `saAmfResponse_4()`.
- **Component-invoked healthcheck**: this variant of healthcheck is invoked by the component itself (according to its configured parameters, see Section 7.1.2.4), and the component reports the result of the healthcheck to the Availability Management Framework by calling `saAmfHealthcheckConfirm()`.

7.1.2.3 Starting and Stopping Healthchecks

Healthchecks are started when a process invokes the `saAmfHealthcheckStart()` function; they are stopped when a process invokes the `saAmfHealthcheckStop()` function. There is no default healthcheck that is invoked by the Availability Management Framework without an explicit start request by the component.

Multiple processes of a component can start healthcheck and each one can decide which healthcheck should be performed. Moreover, when a process starts a healthcheck, it can also specify the recommended recovery action to be applied by the Availability Management Framework when it reports an error on the component if its healthcheck reports to the Availability Management Framework are not made in a timely manner.

The start of healthchecks is independent from the component registration, that is, it is possible to start healthchecks before the component is registered.

7.1.2.4 Healthcheck Configuration Issues

The Availability Management Framework supports the notion of a healthcheck type. A number of healthcheck types can be defined for a component type, and each healthcheck type is identified by a healthcheck key.

A healthcheck type configuration must be provided for each healthcheck key that the component uses to start a healthcheck. All components of the same type share the healthcheck attribute values defined in the healthcheck type configuration.

The healthcheck attributes provided in the healthcheck type for a certain healthcheck key can be overridden in the healthcheck configuration for the component for the

same healthcheck key.
The healthcheck configuration for the component can only specify healthcheck keys for which there is a healthcheck type configuration for its component type.

Further details on the healthcheck configuration are presented in Chapter 8 and [7].

The Availability Management Framework retrieves the healthcheck configuration based on the healthcheck key referred to by the `healthcheckKey` parameter of the healthcheck API calls. The scope of the healthcheck key is limited to the component and is not cluster-wide.

It is assumed that the component configuration retrieved by the Availability Management Framework has passed a series of sanity checks and validations before the cluster startup. Hence, this rules out errors like specifying too frequent healthcheck in the configuration. Thus, based on these validations, the Availability Management Framework may reject a healthcheck start request only if some of the given parameters, such as component name or `healthcheckKey`, are invalid.

A healthcheck configuration comprises two attributes:

- `period`: this attribute indicates the period at which the corresponding healthcheck should be initiated. This attribute is defined for both framework-invoked and component-invoked healthchecks; however, it has different meanings for these two variants of healthchecks, as will be explained next. The name of this configuration attribute is either `saAmfHealthcheckPeriod` (contained in the `SaAmfHealthcheck` configuration object class), if the healthcheck is configured specifically for the component or `saAmfHctDefPeriod` (contained in the `SaAmfHealthcheckType` configuration object class), if the healthcheck is configured for the component type. These configuration object classes are shown in Section 8.14.
- `maximum-duration`: this attribute indicates the time-limit after which the Availability Management Framework will report an error on the component if no response for a healthcheck is received by the Availability Management Framework in this time frame. This attribute applies only to the framework-invoked healthcheck variant. The name of this configuration attribute is either `saAmfHealthcheckMaxDuration` (contained in the `SaAmfHealthcheck` configuration object class), if the healthcheck is configured specifically for the component or `saAmfHctDefMaxDuration` (contained in the `SaAmfHealthcheckType` configuration object class), if the healthcheck is configured for the component type. These configuration object classes are shown in Section 8.14.

The component developer is aware of the healthcheck variant supported by a component, and the component developer specifies this healthcheck variant in the corresponding healthcheck API calls.

7.1.2.4.1 Role of Period and Maximum-Duration in Framework-Invoked Healthchecks

- `period`: for a given framework-invoked healthcheck started by a process and for every "period", the Availability Management Framework will invoke the corresponding healthcheck callback; however, if the process does not respond to a given healthcheck callback before the start of the next healthcheck period, the Availability Management Framework will not trigger the next invocation of the healthcheck callback until the response to the previous invocation is received. In other words, at any given time and for each healthcheck, there is at most one callback invocation for which the response is pending. Of course, as a process may have started several healthchecks in parallel, the Availability Management Framework will invoke callbacks for these different healthchecks independently. How the Availability Management Framework reacts if the process does not respond timely to a framework-invoked healthcheck is explained in the next bullet.
- `maximum-duration`: to correctly specify the value for the period of a healthcheck, the deployer has to make sure that the period is set larger than the average duration of the interval between the Availability Management Framework triggering a callback invocation and receiving the corresponding response. This setting guarantees that in normal conditions, with expected load, the response of a healthy process for the invoked healthcheck callbacks will arrive at the Availability Management Framework timely (and before the Availability Management Framework attempts to issue another callback for the same healthcheck). However, it may not be very easy for the deployer to estimate the expected normal condition and load on the cluster. Therefore, the Availability Management Framework should wait somewhat longer than this average time before concluding that the process is unable to respond to the healthcheck. The `maximum-duration` attribute is defined for such a purpose: the Availability Management Framework will wait for `maximum-duration` to receive a response from the process (component) for a given callback invocation. The deployer should allow enough slack in the `maximum-duration` attribute, so that the response of the healthy process (component) will definitely arrive at the Availability Management Framework before `maximum-duration` expires, even in presence of situations such as high-load on the network and/or high-load on the processing resources of nodes in the cluster.

In short, one has to consider the following trade-off in defining values for `period` and `maximum-duration` for the framework-invoked healthchecks:

- `period`: this value should be set as short as possible, but it should be larger than the average time-duration accounted for the arrival of the corresponding reply to the Availability Management Framework. If `period` is set too short, the Availability Management Framework may consider the component of a healthy process that runs in highly load environment as faulty. On the other hand, if `period` is set too large, the process may be checked too sparsely, and thus the latency in detecting process (component) failures (mostly latent fault detection) becomes larger.
- `maximum-duration`: as discussed earlier, `maximum-duration` should be larger than the average time-duration accounted for the response of a process to a callback invocation. The `maximum-duration` attribute should also include enough slack time, so that, even in the presence of anomalies other than component failures, the healthcheck response arrives at the Availability Management Framework before `maximum-duration` expires. If `maximum-duration` is set too short, it is possible that a healthy process (component) has not been given enough time to respond to the healthcheck. In this case, the Availability Management Framework will falsely assume that the component is faulty. On the other hand, if `maximum-duration` is set too large, the latency for the detection of a faulty component being healthchecked may be increased.

7.1.2.4.2 Role of Period in Component-Invoked Healthchecks

As already explained, component-invoked healthchecks do not have the `maximum-duration` attribute (if it is provided, it will be ignored by the Availability Management Framework). When a process informs the Availability Management Framework of its intention of starting a component-invoked healthcheck (by calling `saAmfHealthcheckStart()`), the Availability Management Framework expects that the process invokes periodically `saAmfHealthcheckConfirm()` no later than at the end of every period. More specifically, the Availability Management Framework reports an error on the component if the Availability Management Framework does not receive a healthcheck confirmation from the component before the end of every period. The recommended recovery for this error is specified by the process when it invoked the `saAmfHealthcheckStart()` call. The deployer should add enough slack time to `period`, so that the healthcheck invoked by a healthy process can reach the Availability Management Framework on time.

7.1.2.4.3 Modification of Healthcheck Parameters

Modifications of the `period` and `maximum-duration` healthcheck attributes for Framework-invoked healthchecks in the Availability Management Framework configuration take place immediately. In contrast, modifications of the `period` healthcheck attribute for component-invoked healthchecks in the Availability Management Framework configuration will be effective the next time the healthcheck is started by invoking the `saAmfHealthcheckStart()` function.

7.1.3 Component Service Instance Management

The basic concepts have been explained in Chapter 3.

Administrative, operational, and presence states are managed by the Availability Management Framework, but they are not exposed to the components. The readiness state of a component is a private state managed by the Availability Management Framework. It is neither exposed to components nor to system management, and it is solely used to determine the eligibility of components to receive component service instance assignments.

The APIs exposed by the Availability Management Framework are limited to the management of the HA state and the HA readiness state for components.

The Availability Management Framework uses callbacks to request components to

- add or remove component service instances from components that are in the in-service state and to
- change the HA state of a component on behalf of a component service instance (active, standby, quiescing, quiesced).

The registered process for a pre-instantiable component uses the `saAmfHAReadinessStateSet()` API function to inform the Availability Management Framework about any change in the HA readiness states of the component for the component service instances that can be assigned to it (see Section 3.2.2.5). The Availability Management Framework must take into account these changes immediately and, if necessary, change the HA state of the component for its assigned component service instances (possibly re-assigning the affected component service instances to other components). The registered process for a pre-instantiable component is responsible for the timely and accurate update of the HA readiness states for the component service instances that can be assigned to it.

The Availability Management Framework enforces that there are no overlapping requests to set the state of a component at any specific time. Two state change requests are said to overlap if the Availability Management Framework requests a component to enter the new state, before the component acknowledges the first request, which is done when the component invokes the `saAmfResponse_4()` API function, as described in Section 7.13.1. The rationale for avoiding overlapping requests is that it is simpler to program a component when overlapping requests are prohibited than when the component must check and report such overlapping.

Component service instances can be assigned to a component only if the component is in the in-service readiness state, and its HA readiness state for the component service instance allows for its assignment in the intended HA state. For details, refer to the readiness state in Section 3.2.2.3, to the HA state in Section 3.2.2.4, and to the HA readiness state in Section 3.2.2.5.

The component service instance management comprises data structures and APIs. The API functions are described in Section 7.9 on page 293.

7.1.4 Component Life Cycle Management

The API functions of the component life cycle management are described in Section 7.10 on page 307. They comprise the callback function to request a component to terminate and the callback functions that proxy and container components export to enable the Availability Management Framework to manage proxied and contained components.

7.1.5 Protection Group Management

The basic concepts have been explained in Chapter 3. For the API functions, refer to Section 7.11 on page 316.

7.1.6 Error Reporting

For the API interfaces, refer to Section 7.12 on page 325.

7.1.7 Correlation of Notifications

Some events such as errors or administrative operations may trigger extensive changes in the cluster and hence generate a large number of related notifications.

These notifications can be organized in a tree structure, the root of which is the notification for the triggering event. To allow a management application to reconstruct this tree, notifications are correlated using their notification identifiers. In case of the Availability Management Framework, this functionality is enabled by including in the error reporting APIs the data structure defined by the SA Forum Notification Service ([3]) for this purpose.

Whenever applicable, it is recommended that a process reporting an error condition, for which one or more notifications have already been generated, includes as correlated notifications the identifiers of the root and parent notifications within the notification tree structure.
The Availability Management Framework will generate an error report notification as a result of the error report and use any correlated notifications the caller process indicated. The Availability Management Framework will also return the identifier of the generated error report notification to the caller.

When a process invokes the API to clear the error condition, it shall provide the root and parent notification identifiers that connect the current action to the tree of other related events.
As a result of the invocation, the Availability Management Framework will generate an error clear notification which includes these two notification identifiers along with all identifiers of error report notifications previously sent for the error being cleared. The Availability Management Framework will also return the identifier of the generated error clear notification to the caller process.

Applications may generate notifications on their own as a result of different callbacks received from the Availability Management Framework. Each of these callbacks is identified by an `invocation` parameter. The invoked process may invoke the `saAmfCorrelationIdsGet()` API function (see Section 7.12.3) with this parameter to obtain any relevant correlated notification identifiers for the event that triggered the particular Availability Management Framework's callback request.

7.1.8 Component Response to Framework Requests

For the API interfaces, refer to Section 7.13 on page 333.

7.1.9 API Usage Illustrations

This section illustrates the usage of the Availability Management Framework API by different categories of components.

FIGURE 24 shows an example of an SA-aware component consisting of a single process. The numbers in circles indicate the sequence of events in time.

FIGURE 24 SA-Aware Component Consisting of a Single Process

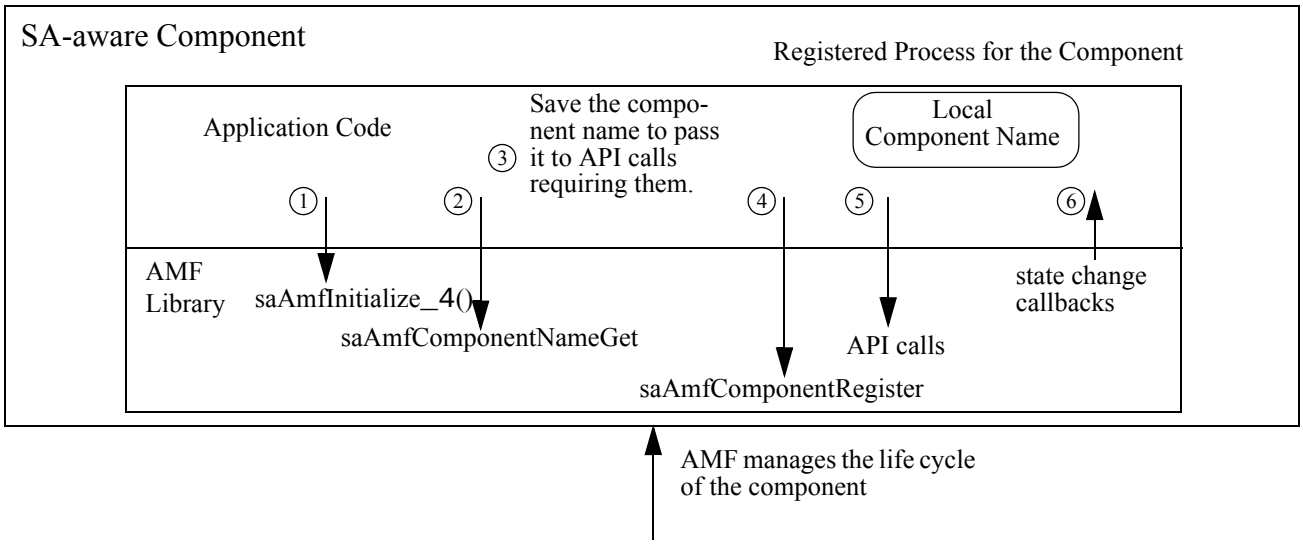


FIGURE 25 shows an example of an SA-aware component consisting of multiple processes. The numbers in circles indicate the sequence of events in time.

FIGURE 25 SA-Aware Component Consisting of Multiple Processes

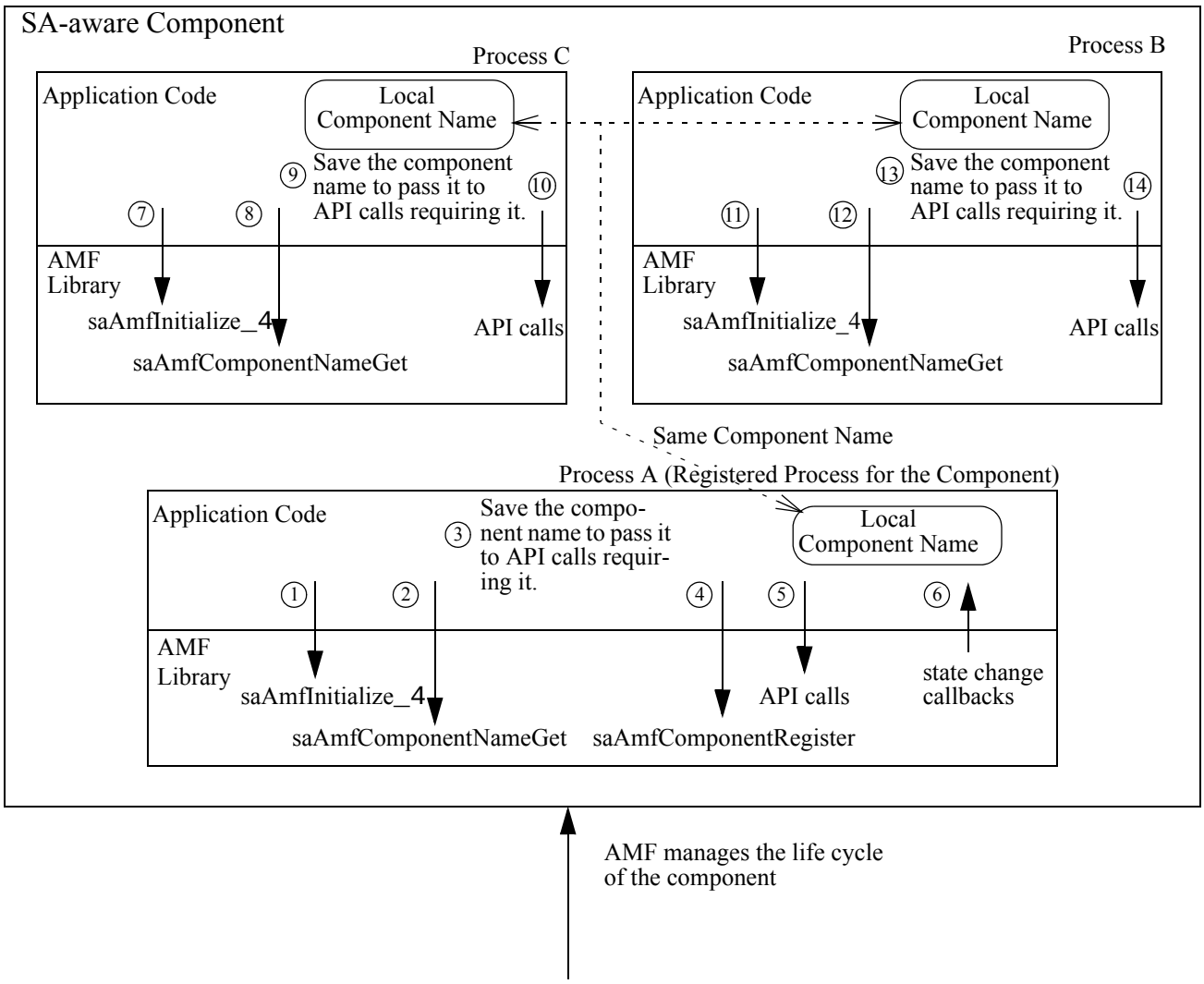
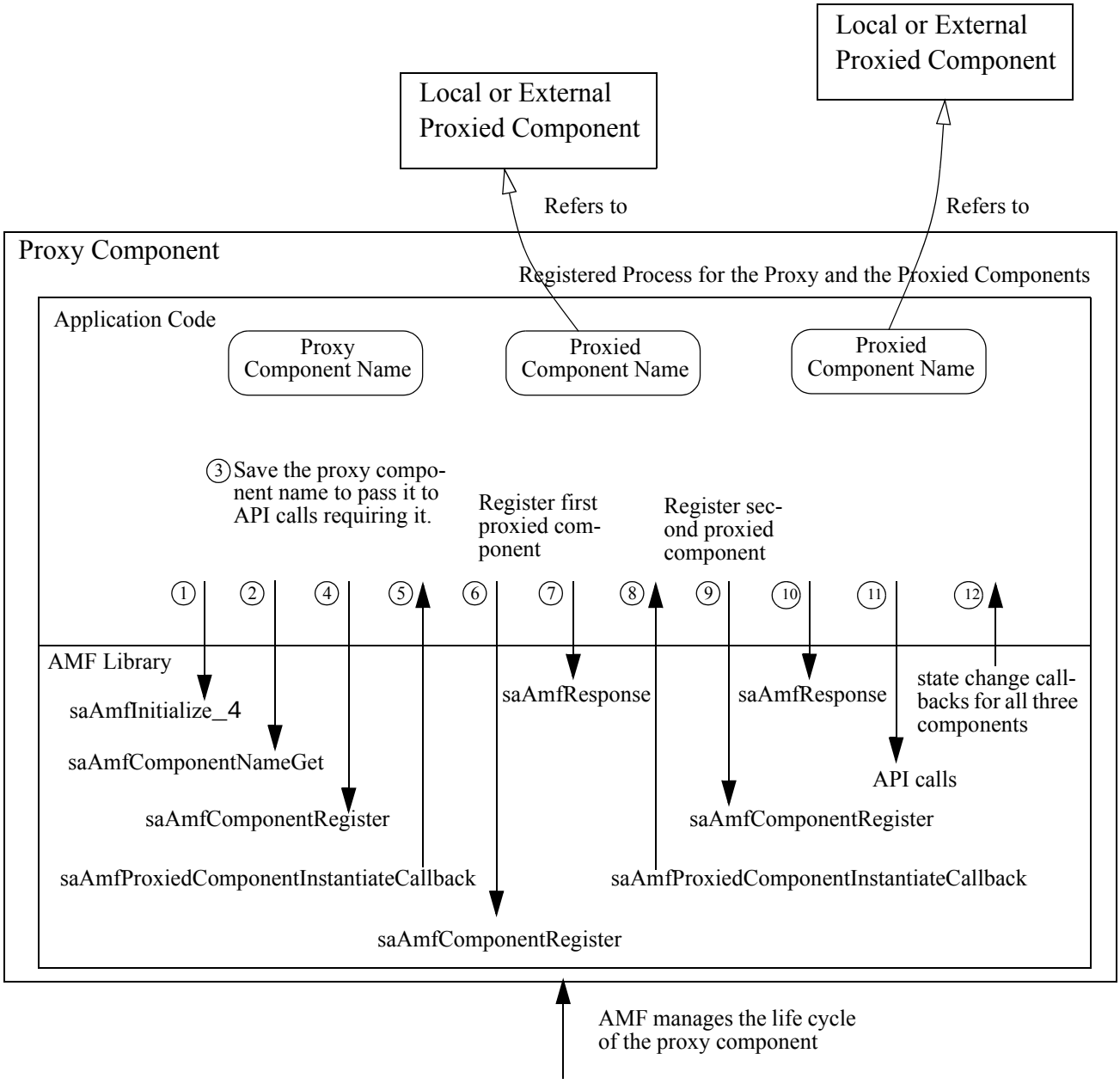


FIGURE 26 shows an example of a single-process proxy component that registers itself (in step 4) and two proxied components with the Availability Management Framework. The proxy component registers the proxied components (in steps 6 and 9) before it replies to the callbacks to instantiate the proxied components (in steps 7 and 10). The numbers in circles indicate the sequence of events in time.

FIGURE 26 A Single-Process Proxy Component and Two Proxied Components



7.2 Unavailability of the AMF API on a Non-Member Node

The following subsection describes the behavior of the Availability Management Framework under various conditions that cause the Availability Management Framework to be unavailable on a node. Section 7.2.2 contains guidelines for Availability Management Framework implementers for dealing with a temporary unavailability of the service.

7.2.1 A Member Node Leaves or Rejoins the Cluster Membership

As described in Section 3.1.1.1, the Availability Management Framework does not provide service to processes on cluster nodes that are not in the cluster membership (see [4]).

The Availability Management Framework is notified by the Cluster Membership Service about cluster membership changes. How the Availability Management Framework reacts to these changes is explained in detail in Appendix D. In this section, only the behavior from the perspective of a process is described.

When the Availability Management Framework detects that the CLM node to which an AMF node is mapped has unexpectedly left the cluster membership, the Availability Management Framework abruptly terminates all components hosted by this AMF node by executing the CLEANUP CLC-CLI command (see Section 4.8) for all its local components. Thus, when a node has left the cluster membership, no processes belonging to components should be running on the node. However, there are a few special situations in which processes may call Availability Management Framework API functions.

- An Availability Management Framework API function is called by a process nearly at the same time when the node exits the cluster and the Availability Management Framework area server on the node has not yet terminated the process.
- The Availability Management Framework encounters an error when attempting to terminate all of the processes belonging to components, so there may still be processes running.
- The cleanup operation of a component (see Table 37 in Appendix A) does not properly terminate all its processes.
- A process using the Availability Management Framework API, but which is not part of a component, is running on the AMF node, and since the Availability Management Framework has no knowledge of the process, the Availability Management Framework will not attempt to clean it up.

In the few special situations described above, the Availability Management Framework behaves as follows towards processes residing on that node and using or attempting to use the service:

- Calls to `saAmfInitialize_4()` will fail with `SA_AIS_ERR_UNAVAILABLE`.
- All Availability Management Framework APIs that are invoked by the process and that operate on handles already acquired by the process will fail with `SA_AIS_ERR_UNAVAILABLE` with the exception of `saAmfFinalize()`, which is used to free the library handles and all resources associated with these handles.
- Any outstanding `SaAmfProtectionGroupTrackCallbackT_4` callback will provide `SA_AIS_ERR_UNAVAILABLE` in the error parameter.
- No other callbacks will be called.

If the node rejoins the cluster membership, the Availability Management Framework instantiates service units on this node based on the configuration of the service groups that contain service units hosted by that node. Processes belonging to components of these service units can access the Availability Management Framework API functions without restrictions. However, the left-over processes of the few special situations above will still be denied service as explained.

When the node leaves the membership, the Availability Management Framework executing on the remaining nodes of the cluster behaves as if all processes belonging to components residing on the leaving node had been terminated.

As AMF engages procedures to terminate all components on the leaving node, AMF sets the presence state of all components and all service units on this node to uninstantiated. The readiness state of all service units and all components on this node is set to out-of-service.

7.2.2 Guidelines for Availability Management Framework Implementers

The implementation of the Availability Management Framework must leverage the SA Forum Cluster Membership Service (see [4]) to determine the membership status of a node. If the Cluster Membership Service considers a node as a member of the cluster but the Availability Management Framework experiences difficulty in providing service to its clients because of transport, communication, or other issues, it must respond to the API calls invoked by a process with `SA_AIS_ERR_TRY_AGAIN`.

7.3 Include File and Library Names

The following statements containing declarations of data types and function prototypes must be included in the source of an application using the Availability Management Framework API:

```
#include <saAmf.h>
```

and

```
#include <saNtf.h>
```

The latter statement is needed for the functions `saAmfComponentErrorReport_4()`, `saAmfComponentErrorClear_4()`, `saAmfCorrelationIdsGet()`, `saAmfHAReadinessStateSet()`, and `saAmfResponse_4()`.

To use the Availability Management Framework API, an application must be bound with the following library:

```
libSaAmf.so
```

7.4 Type Definitions

The Availability Management Framework uses the types described in the following sections.

7.4.1 SaAmfHandleT

```
typedef SaUint64T SaAmfHandleT;
```

A process acquires this handle to the Availability Management Framework by invoking the `saAmfInitialize_4()` function and uses it in subsequent invocations of the functions of the Availability Management Framework.

7.4.2 Component Process Monitoring

This section describes the data types that the Availability Management Framework requires for the passive monitoring of processes of a component.

7.4.2.1 SaAmfPmErrorsT Type

```
#define SA_AMF_PM_ZERO_EXIT          0x1
#define SA_AMF_PM_NON_ZERO_EXIT      0x2
#define SA_AMF_PM_ABNORMAL_END       0x4
typedef SaUint32T SaAmfPmErrorsT;
```

7.4.2.2 SaAmfPmStopQualifierT Type

```
typedef enum {
     SA_AMF_PM_PROC                        = 1,
     SA_AMF_PM_PROC_AND_DESCENDENTS        = 2,
     SA_AMF_PM_ALL_PROCESSES               = 3
} SaAmfPmStopQualifierT;
```

For the explanation of the enum values in `SaAmfPmStopQualifierT`, refer to Section 7.7.2 on page 280.

7.4.3 Component Healthcheck Monitoring

7.4.3.1 SaAmfHealthcheckInvocationT

```
typedef enum {
     SA_AMF_HEALTHCHECK_AMF_INVOKED              = 1,
     SA_AMF_HEALTHCHECK_COMPONENT_INVOKED        = 2
} SaAmfHealthcheckInvocationT;
```

The values of the `SaAmfHeathcheckInvocationT` enumeration type are:

- `SA_AMF_HEALTHCHECK_AMF_INVOKED` - The healthchecks are invoked by the Availability Management Framework.
- `SA_AMF_HEALTHCHECK_COMPONENT_INVOKED` - The healthchecks are invoked by the component.

7.4.3.2 SaAmfHealthcheckKeyT

```
#define SA_AMF_HEALTHCHECK_KEY_MAX 32

typedef struct {
     SaUint8T key[SA_AMF_HEALTHCHECK_KEY_MAX];
     SaUint16T keyLen;
} SaAmfHealthcheckKeyT;
```

7.4.4 Types for State Management

7.4.4.1 HA State

```
typedef enum {
     SA_AMF_HA_ACTIVE          = 1,
     SA_AMF_HA_STANDBY         = 2,
     SA_AMF_HA_QUIESCED        = 3,
     SA_AMF_HA_QUIESCING       = 4
} SaAmfHAStateT;
```

The HA state is active, standby, quiesced, or quiescing.

7.4.4.2 Readiness State

```
typedef enum {
     SA_AMF_READINESS_OUT_OF_SERVICE       = 1,
     SA_AMF_READINESS_IN_SERVICE           = 2,
     SA_AMF_READINESS_STOPPING             = 3
} SaAmfReadinessStateT;
```

The readiness state is out-of-service, in-service, or stopping.

7.4.4.3 Presence State

```
typedef enum {
     SA_AMF_PRESENCE_UNINSTANTIATED            = 1,
     SA_AMF_PRESENCE_INSTANTIATING             = 2,
     SA_AMF_PRESENCE_INSTANTIATED              = 3,
     SA_AMF_PRESENCE_TERMINATING               = 4,
     SA_AMF_PRESENCE_RESTARTING                = 5,
     SA_AMF_PRESENCE_INSTANTIATION_FAILED      = 6,
     SA_AMF_PRESENCE_TERMINATION_FAILED        = 7
} SaAmfPresenceStateT;
```

The presence state is uninstantiated, instantiating, instantiated, terminating, restarting, instantiation-failed, or termination-failed.

7.4.4.4 Operational State

```
typedef enum {
     SA_AMF_OPERATIONAL_ENABLED      = 1,
     SA_AMF_OPERATIONAL_DISABLED     = 2
} SaAmfOperationalStateT;
```

The operational state is enabled or disabled.

7.4.4.5 Administrative State

```
typedef enum {
     SA_AMF_ADMIN_UNLOCKED               = 1,
     SA_AMF_ADMIN_LOCKED                 = 2,
     SA_AMF_ADMIN_LOCKED_INSTANTIATION   = 3,
     SA_AMF_ADMIN_SHUTTING_DOWN          = 4
} SaAmfAdminStateT;
```

The administrative state is unlocked, locked, locked-instantiation, or shutting-down.

7.4.4.6 Assignment State

```
typedef enum {
     SA_AMF_ASSIGNMENT_UNASSIGNED              = 1,
     SA_AMF_ASSIGNMENT_FULLY_ASSIGNED          = 2,
     SA_AMF_ASSIGNMENT_PARTIALLY_ASSIGNED      = 3
} SaAmfAssignmentStateT;
```

The assignment state of an SI is unassigned, fully-assigned, or partially-assigned.

7.4.4.7 HA Readiness State

```
typedef enum {
     SA_AMF_HARS_READY_FOR_ASSIGNMENT      = 1,
     SA_AMF_HARS_READY_FOR_ACTIVE_DEGRADED = 2,
     SA_AMF_HARS_NOT_READY_FOR_ACTIVE      = 3,
     SA_AMF_HARS_NOT_READY_FOR_ASSIGNMENT  = 4
} SaAmfHAReadinessStateT;
```

The HA readiness state is ready-for-assignment, ready-for-active-degraded, not-ready-for-active, or not-ready-for-assignment.

7.4.4.8 Proxy Status

```
typedef enum {
     SA_AMF_PROXY_STATUS_UNPROXIED         = 1,
     SA_AMF_PROXY_STATUS_PROXIED           = 2
} SaAmfProxyStatusT;
```

The proxy status of a component is proxied (SA_AMF_PROXY_STATUS_PROXIED) or unproxied (SA_AMF_PROXY_STATUS_UNPROXIED). If the proxy status is SA_AMF_PROXY_STATUS_PROXIED, a proxy component is currently “proxying” the component. If the proxy status is SA_AMF_PROXY_STATUS_UNPROXIED, no proxy component is currently assigned to “proxy” the component, possibly because the previous proxy component failed, and the Availability Management Framework could not engage another component to assume the mediation responsibility for the component.

7.4.4.9 All Defined States

```
typedef enum {
     SA_AMF_READINESS_STATE     = 1,
     SA_AMF_HA_STATE            = 2,
     SA_AMF_PRESENCE_STATE      = 3,
     SA_AMF_OP_STATE            = 4,
     SA_AMF_ADMIN_STATE         = 5,
     SA_AMF_ASSIGNMENT_STATE    = 6,
     SA_AMF_PROXY_STATUS        = 7,
     SA_AMF_HA_READINESS_STATE  = 8
} SaAmfStateT;
```

This enum defines all states (readiness, HA state, presence, operational, administrative, assignment, and HA readiness) and the additional proxy status.

7.4.5 Component Service Types

7.4.5.1 SaAmfCSIFlagsT

```
#define SA_AMF_CSI_ADD_ONE       0X1
#define SA_AMF_CSI_TARGET_ONE    0X2
#define SA_AMF_CSI_TARGET_ALL    0X4
typedef SaUint32T SaAmfCSIFlagsT;
```

The values for the `SaAmfCSIFlagsT` are the following:

- `SA_AMF_CSI_ADD_ONE` - A new component service instance is assigned to the component. The component is requested to assume a particular HA state for the new component service instance.
- `SA_AMF_CSI_TARGET_ONE` - The request made to the component targets only one of its component service instances.
- `SA_AMF_CSI_TARGET_ALL` - The request made to the component targets all of its component service instances. This flag is used for cases in which all component service instances are managed as a bundle: the component is assigned the same HA state for all component service instances at the same time, or all component service instances are removed at the same time. This flag is used for removing all component service instances at once, if it makes sense.

These values are mutually exclusive. Only one value can be set in SaAmfCSIFlagsT.

7.4.5.2 SaAmfCSITransitionDescriptorT

```
typedef enum {
     SA_AMF_CSI_NEW_ASSIGN      = 1,
     SA_AMF_CSI_QUIESCED        = 2,
     SA_AMF_CSI_NOT_QUIESCED    = 3,
     SA_AMF_CSI_STILL_ACTIVE    = 4
} SaAmfCSITransitionDescriptorT;
```

This enumeration type provides information on the component that was or still is active for the specified component service instance. The values of the SaAmfCSITransitionDescriptorT enumeration type have the following interpretation:

- SA_AMF_CSI_NEW_ASSIGN - This assignment is not the result of a switch-over or fail-over of the specified component service instance from another component to this component. No component was previously active for this component service instance.
- SA_AMF_CSI_QUIESCED - This assignment is the result of a switch-over of the specified component service instance from another component to this component. The component that was previously active for this component service instance has been quiesced.
- SA_AMF_CSI_NOT_QUIESCED - This assignment is the result of a fail-over of the specified component service instance from another component to this component. The component that was previously active for this component service instance has not been quiesced.
- SA_AMF_CSI_STILL_ACTIVE - This value is only used in the N-way active redundancy model when the assignment is not the result of a switchover or a failover of the specified component service instance from another component to this component, and at least one other component is already assigned active for that component service instance.

7.4.5.3 SaAmfCSIStateDescriptorT

```
typedef struct {
      SaAmfCSITransitionDescriptorT transitionDescriptor;
      SaNameT activeCompName;
} SaAmfCSIActiveDescriptorT;
```

The fields of the `SaAmfCSIActiveDescriptorT` structure have the following interpretation:

- `transitionDescriptor` - This descriptor provides information on the component that was or is still active for the one or all of the specified component service instances (see previous section).
- `activeCompName` - The name of the component that was previously active for the specified component service instance.

When a component is requested to assume the active HA state for one or for all component service instances assigned to the component, `SaAmfCSIActiveDescriptorT` holds the following information:

- The Availability Management Framework uses the `transitionDescriptor` that is appropriate for the redundancy model of the service group to which this component belongs.
- If `transitionDescriptor` is set to `SA_AMF_CSI_NOT_QUIESCED` or `SA_AMF_CSI_QUIESCED`, `activeCompName` holds the name of the component that was previously assigned the active state for the component service instances and no longer has that assignment.
- If `transitionDescriptor` is set to `SA_AMF_CSI_NEW_ASSIGN`, `activeCompName` is not used.
- If `transitionDescriptor` is set to `SA_AMF_CSI_STILL_ACTIVE`, `activeCompName` holds the name of one of the components that are still assigned the active HA state for all targeted component service instances. Any of these components can be arbitrarily selected.

```
typedef struct {
      SaNameT activeCompName;
      SaUint32T standbyRank;
} SaAmfCSIStandbyDescriptorT;
```

The fields of the SaAmfCSIStandbyDescriptorT structure have the following interpretation:

- activeCompName - Name of the component that is currently active for the one or all of the specified component service instances. This name is empty if no active component exists.
- standbyRank - Rank of the component for assignments of the standby HA state to the component for the one or all of the specified component service instances.

When a component is requested to assume the standby HA state for one or for all component service instances assigned to the component, SaAmfCSIStandbyDescriptorT holds in activeCompName the name of the component that is currently assigned the active state for the one or all these component service instances. In redundancy models in which several components may assume the standby HA state for the same component service instance at the same time, standbyRank indicates to the component the rank it must assume. When the Availability Management Framework selects a component to assume the active HA state for a component service instance, the component assuming the standby state for that component service instance with the lowest standbyRank value is chosen.

```
typedef union {
     SaAmfCSIActiveDescriptorT activeDescriptor;
     SaAmfCSIStandbyDescriptorT standbyDescriptor;
} SaAmfCSIStateDescriptorT;
```

The SaAmfCSIStateDescriptorT holds additional information about the assignment of a component service instance to a component when the component is requested to assume the active or standby HA state for this component service instance.

7.4.5.4 SaAmfCSIAttributeListT

```
typedef struct {
     SaUint8T *attrName;
     SaUint8T *attrValue;
} SaAmfCSIAttributeT;
```

SaAmfCSIAttributeT represents a single component service instance attribute by its name and value strings. Each string consists of UTF-8 encoded characters and is terminated by the NULL character.

```
typedef struct {
     SaAmfCSIAttributeT *attr;
     SaUint32T number;
} SaAmfCSIAttributeListT;
```

SaAmfCSIAttributeListT represents the list of all attributes of a single component service instance. The attr pointer points to an array of number elements of SaAmfCSIAttributeT attribute descriptors.

7.4.5.5 SaAmfCSIDescriptorT

```
typedef struct {
     SaAmfCSIFlagsT csiFlags;
     SaNameT csiName;
     SaAmfCSIStateDescriptorT csiStateDescriptor;
     SaAmfCSIAttributeListT csiAttr;
} SaAmfCSIDescriptorT;
```

SaAmfCSIDescriptorT provides information about the component service instances targeted by the saAmfCSISetCallback() callback API.

When SA_AMF_CSI_TARGET_ALL is set in csiFlags, csiName is not used; otherwise, csiName contains the name of the component service instance targeted by the callback.

When SA_AMF_CSI_ADD_ONE is set in csiFlags, csiAttr refers to the attributes of the newly assigned component service instance; otherwise, no attributes are provided, and csiAttr is not used.

When the component is requested to assume the active or standby state for the targeted service instances, csiStateDescriptor holds additional information relative to that state transition; otherwise, csiStateDescriptor is not used.

7.4.6 Types for Protection Group Management

7.4.6.1 SaAmfProtectionGroupMemberT_4

```
typedef struct {
      SaNameT compName;
      SaAmfHAStateT haState;
      SaAmfHAReadinessStateT haReadinessState;
      SaUint32T rank;
} SaAmfProtectionGroupMemberT_4;
```

The fields of the `SaAmfProtectionGroupMemberT_4` structure have the following interpretation:

- `compName` - The name of the component that is a member of the protection group.
- `haState` - The HA state of the member component for the component service instance supported by the member component.
- `haReadinessState` - The HA readiness state of the member component for the component service instance protected by the protection group.
- `rank` - The standby rank of the member component in the protection group if `haState` is standby.

7.4.6.2 SaAmfProtectionGroupChangesT

```
typedef enum {
      SA_AMF_PROTECTION_GROUP_NO_CHANGE          = 1,
      SA_AMF_PROTECTION_GROUP_ADDED              = 2,
      SA_AMF_PROTECTION_GROUP_REMOVED            = 3,
      SA_AMF_PROTECTION_GROUP_STATE_CHANGE       = 4
} SaAmfProtectionGroupChangesT;
```

The values of the `SaAmfProtectionGroupChangesT` enumeration type have the following interpretation:

- SA_AMF_PROTECTION_GROUP_NO_CHANGE - This value is used when the `trackFlags` parameter of the `saAmfProtectionGroupTrack_4()` function (as defined in Section 7.11.1) is either

  ⇒ SA_TRACK_CURRENT or

  ⇒ SA_TRACK_CHANGES, and all the following conditions hold:

- The member component was already a member of the protection group in the previous `saAmfProtectionGroupTrackCallback()` callback call.
- The component service instance has not been removed from the member component.
- Neither `haState`, `haReadinessState`, nor `rank` of the `SaAmfProtectionGroupMemberT_4` structure of this member component has changed.

- `SA_AMF_PROTECTION_GROUP_ADDED` - The associated component service instance has been added to the member component.
- `SA_AMF_PROTECTION_GROUP_REMOVED` - The associated component service instance has been removed from the member component.
- `SA_AMF_PROTECTION_GROUP_STATE_CHANGE` - Any of the elements `haState`, `haReadinessState`, or `rank` of the `SaAmfProtectionGroupMemberT_4` structure for the member component have changed.

7.4.6.3 SaAmfProtectionGroupNotificationT_4

```
typedef struct {
     SaAmfProtectionGroupMemberT_4 member;
     SaAmfProtectionGroupChangesT change;
} SaAmfProtectionGroupNotificationT_4;
```

The fields of the `SaAmfProtectionGroupNotificationT_4` structure have the following interpretation:

- `member` - The information associated with the component member of the protection group.
- `change` - The kind of change in the associated component member.

7.4.6.4 SaAmfProtectionGroupNotificationBufferT_4

```
typedef struct {
     SaUint32T numberOfItems;
     SaAmfProtectionGroupNotificationT_4 *notification;
} SaAmfProtectionGroupNotificationBufferT_4;
```

The fields of the `SaAmfProtectionGroupNotificationBufferT_4` structure have the following interpretation:

- `numberOfItems` - Number of elements of type `SaAmfProtectionGroupNotificationT_4` in the array to which `notification` points.
- `notification` - Pointer to the notification array.

7.4.7 SaAmfRecommendedRecoveryT

```
typedef enum {
     SA_AMF_NO_RECOMMENDATION         = 1,
     SA_AMF_COMPONENT_RESTART         = 2,
     SA_AMF_COMPONENT_FAILOVER        = 3,
     SA_AMF_NODE_SWITCHOVER           = 4,
     SA_AMF_NODE_FAILOVER             = 5,
     SA_AMF_NODE_FAILFAST             = 6,
     SA_AMF_CLUSTER_RESET             = 7,
     SA_AMF_APPLICATION_RESTART       = 8,
     SA_AMF_CONTAINER_RESTART         = 9
} SaAmfRecommendedRecoveryT;
```

A short explanation of the values of this enumeration is given next. Additional details are provided in Section 3.11.1.3 and subsections:

- `SA_AMF_NO_RECOMMENDATION` - This report makes no recommendation for recovery. However, the Availability Management Framework should engage the configured per-component recovery policy (refer to Section 3.11.1.3) in such a scenario.
- `SA_AMF_COMPONENT_RESTART` - The erroneous component should be terminated and reinstantiated.
- `SA_AMF_COMPONENT_FAILOVER` - The error is related to the execution environment of the component on the current node. Depending on the redundancy model used, either the component or the service unit containing the component should fail over to another node.
- `SA_AMF_NODE_SWITCHOVER` - The error has been identified as being at the node level, and no service instance should be assigned to service units on that node. Service instances containing component service instances assigned to the failed component are failed over while other service instances are switched over to other nodes (component service instances are not abruptly removed; instead, they are brought to the quiesced state before being removed).

- SA_AMF_NODE_FAILOVER - The error has been identified as being at the node level, and no service instance should be assigned to service units on that node. All service instances assigned to service units contained in the node are failed over to other nodes (by an abrupt termination of all node-local components).
- SA_AMF_NODE_FAILFAST - The error has been identified as being at the node level, and components should not be in service on the node. The node should be rebooted using a low-level interface.
- SA_AMF_APPLICATION_RESTART - The application should be completely terminated and then started again by first terminating all of its service units and then starting them again, ensuring that—during the termination phase of the restart procedure—service instances of the application are not reassigned (refer additionally to Section 9.4.7 on page 383).
- SA_AMF_CLUSTER_RESET - The cluster should be reset. In order to execute this function, the Availability Management Framework reboots all nodes that are part of the cluster by using a low level interface without trying to terminate the components individually. To be effective, this operation must be performed such that all AMF nodes are first terminated before any of the AMF nodes starts to instantiate again.
- SA_AMF_CONTAINER_RESTART - Terminate all contained components and the container component abruptly and then instantiate them again.

7.4.8 SaAmfCompCategoryT

```
#define SA_AMF_COMP_SA_AWARE     0x0001
#define SA_AMF_COMP_PROXY        0x0002
#define SA_AMF_COMP_PROXIED      0x0004
#define SA_AMF_COMP_LOCAL        0x0008
#define SA_AMF_COMP_CONTAINER    0x0010
#define SA_AMF_COMP_CONTAINED    0x0020
#define SA_AMF_COMP_PROXIED_NPI  0x0040
typedef SaUint32T SaAmfCompCategoryT;
```

Based on Table 3 on page 50, all possible “ORing” of values are shown in the following table:

Table 20 Possible Combinations of Values in SaAmfCompCategoryT

Component	Mandatory Values	Optional Values
regular SA-aware	SA_AMF_COMP_SA_AWARE	SA_AMF_COMP_LOCAL
proxy, non-container	SA_AMF_COMP_PROXY	SA_AMF_COMP_LOCAL SA_AMF_COMP_SA_AWARE
container, non-proxy	SA_AMF_COMP_CONTAINER	SA_AMF_COMP_LOCAL, SA_AMF_COMP_SA_AWARE
container, proxy	SA_AMF_COMP_CONTAINER. and SA_AMF_COMP_PROXY	SA_AMF_COMP_LOCAL, SA_AMF_COMP_SA_AWARE
contained	SA_AMF_COMP_CONTAINED	SA_AMF_COMP_LOCAL, SA_AMF_COMP_SA_AWARE
local, non-SA-aware, proxied, pre-instantiable	SA_AMF_COMP_LOCAL and SA_AMF_COMP_PROXIED	-
local, non-SA-aware, proxied, non-pre-instantiable	SA_AMF_COMP_LOCAL and SA_AMF_COMP_PROXIED_NPI	-
local, non-SA-aware, non-proxied	SA_AMF_COMP_LOCAL	-
external, pre-instantiable	-	SA_AMF_COMP_PROXIED
external, non-pre-instantiable	SA_AMF_COMP_PROXIED_NPI	-

Components with the following category values are non-pre-instantiable components:

- SA_AMF_COMP_LOCAL (local, non-proxied, non-SA-aware)
- SA_AMF_COMP_LOCAL and SA_AMF_COMP_PROXIED_NPI (local, non-pre-instantiable proxied)
- SA_AMF_COMP_PROXIED_NPI (external, non-pre-instantiable proxied)

All other valid component category values represent pre-instantiable components.

7.4.9 SaAmfRedundancyModelT

```
typedef enum {
     SA_AMF_2N_REDUNDANCY_MODEL                = 1,
     SA_AMF_NPM_REDUNDANCY_MODEL               = 2,
     SA_AMF_N_WAY_REDUNDANCY_MODEL             = 3,
     SA_AMF_N_WAY_ACTIVE_REDUNDANCY_MODEL      = 4,
     SA_AMF_NO_REDUNDANCY_MODEL                = 5
} SaAmfRedundancyModelT;
```

For a description of the various redundancy models enumerated in this type, refer to Section 3.6 on page 109.

7.4.10 SaAmfCompCapabilityModelT

```
typedef enum {
     SA_AMF_COMP_X_ACTIVE_AND_Y_STANDBY          = 1,
     SA_AMF_COMP_X_ACTIVE_OR_Y_STANDBY           = 2,
     SA_AMF_COMP_ONE_ACTIVE_OR_Y_STANDBY         = 3,
     SA_AMF_COMP_ONE_ACTIVE_OR_ONE_STANDBY       = 4,
     SA_AMF_COMP_X_ACTIVE                        = 5,
     SA_AMF_COMP_1_ACTIVE                        = 6,
     SA_AMF_COMP_NON_PRE_INSTANTIABLE            = 7
} SaAmfCompCapabilityModelT;
```

For a description of the values shown in this enum, refer to Section 3.5 on page 107.

7.4.11 Notifications-Related Types

7.4.11.1 SaAmfNotificationMinorIdT

```
typedef enum {
     /* alarms */
     SA_AMF_NTFID_COMP_INSTANTIATION_FAILED      = 0x02,
     SA_AMF_NTFID_COMP_CLEANUP_FAILED            = 0x03,
     SA_AMF_NTFID_CLUSTER_RESET                  = 0x04,
     SA_AMF_NTFID_SI_UNASSIGNED                  = 0x05,
     SA_AMF_NTFID_COMP_UNPROXIED                 = 0x06,

     /* state change */
     SA_AMF_NTFID_NODE_ADMIN_STATE               = 0x065,
     SA_AMF_NTFID_SU_ADMIN_STATE                 = 0x066,
     SA_AMF_NTFID_SG_ADMIN_STATE                 = 0x067,
     SA_AMF_NTFID_SI_ADMIN_STATE                 = 0x068,
     SA_AMF_NTFID_APP_ADMIN_STATE                = 0x069,
     SA_AMF_NTFID_CLUSTER_ADMIN_STATE            = 0x06A,
     SA_AMF_NTFID_NODE_OP_STATE                  = 0x06B,
     SA_AMF_NTFID_SU_OP_STATE                    = 0x06C,
     SA_AMF_NTFID_SU_PRESENCE_STATE              = 0x06D,
     SA_AMF_NTFID_SU_SI_HA_STATE                 = 0x06E,
     SA_AMF_NTFID_SI_ASSIGNMENT_STATE            = 0x06F,
     SA_AMF_NTFID_COMP_PROXY_STATUS              = 0x070,
     SA_AMF_NTFID_SU_SI_HA_READINESS_STATE       = 0x071,

     /* miscellaneous */
     SA_AMF_NTFID_ERROR_REPORT                   = 0x0191,
     SA_AMF_NTFID_ERROR_CLEAR                    = 0x0192
} SaAmfNotificationMinorIdT;
```

This type provides the values for the `minorId` field of notification class identifiers used in notifications of the Availability Management Framework.

7.4.11.2 SaAmfAdditionalInfoIdT

```
typedef enum {
     SA_AMF_NODE_NAME                    = 1,
     SA_AMF_SI_NAME                      = 2,
     SA_AMF_MAINTENANCE_CAMPAIGN_DN      = 3,
     SA_AMF_AI_RECOMMENDED_RECOVERY      = 4,
     SA_AMF_AI_APPLIED_RECOVERY          = 5
}SaAmfAdditionalInfoIdT;
```

The preceding types are used in Availability Management Framework alarms and notifications (refer to Chapter 11) to convey additional information elements in the “Additional Information” field associated with alarms and notifications.

7.4.12 SaAmfCallbacksT_4

```
typedef struct {
     SaAmfHealthcheckCallbackT
          saAmfHealthcheckCallback;
     SaAmfComponentTerminateCallbackT
          saAmfComponentTerminateCallback;
     SaAmfCSISetCallbackT
          saAmfCSISetCallback;
     SaAmfCSIRemoveCallbackT
          saAmfCSIRemoveCallback;
     SaAmfProtectionGroupTrackCallbackT_4
          saAmfProtectionGroupTrackCallback;
     SaAmfProxiedComponentInstantiateCallbackT
          saAmfProxiedComponentInstantiateCallback;
     SaAmfProxiedComponentCleanupCallbackT
          saAmfProxiedComponentCleanupCallback;
     SaAmfContainedComponentInstantiateCallbackT
          saAmfContainedComponentInstantiateCallback;
     SaAmfContainedComponentCleanupCallbackT
          saAmfContainedComponentCleanupCallback;
} SaAmfCallbacksT_4;
```

The `SaAmfCallbacksT_4` structure defines the various callback functions that the Availability Management Framework may invoke on a component.

7.5 Library Life Cycle

7.5.1 saAmfInitialize_4()

Prototype

```
SaAisErrorT saAmfInitialize_4(
      SaAmfHandleT *amfHandle,
      const SaAmfCallbacksT_4 *amfCallbacks,
      SaVersionT *version
);
```

Parameters

`amfHandle` - [`out`] A pointer to the handle which identifies this particular initialization of the Availability Management Framework, and which is to be returned by the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`amfCallbacks` - [`in`] If `amfCallbacks` is set to NULL, no callbacks are registered; if `amfCallbacks` is not set to NULL, it is a pointer to an `SaAmfCallbacksT_4` structure which contains the callback functions of the process that the Availability Management Framework may invoke. Only non-NULL callback functions in this structure will be registered. The `SaAmfCallbacksT_4` is defined in Section 7.4.12 on page 263.

`version` - [`in/out`] As an input parameter, `version` is a pointer to a structure containing the required Availability Management Framework version. In this case, `minorVersion` is ignored and should be set to 0x00.
As an output parameter, `version` is a pointer to a structure containing the version actually supported by the Availability Management Framework. The `SaVersionT` type is defined in [2].

Description

This function initializes the Availability Management Framework for the invoking process and registers the various callback functions. This function must be invoked prior to the invocation of any other Availability Management Framework API function. The handle pointed to by `amfHandle` is returned by the Availability Management Framework as the reference to this association between the process and the Availability Management Framework. The process uses this handle in subsequent communication with the Availability Management Framework.

If the invoking process exits after having successfully returned from the `saAmfInitialize_4()` function and before invoking `saAmfFinalize()` to finalize the handle `amfHandle` (see Section 7.5.4 on page 270), the Availability Management Framework automatically finalizes this handle when it detects the death of the process.

The `amfCallbacks` parameter points to a structure the contains the callbacks that the Availability Management Framework can invoke.

If the implementation supports the version of the Availability Management Framework API specified by the `releaseCode` and `majorVersion` fields of the structure pointed to by the `version` parameter, `SA_AIS_OK` is returned. In this case, the structure pointed to by the `version` parameter is set by this function to:

- `releaseCode` = required release code
- `majorVersion` = highest value of the major version that this implementation can support for the required `releaseCode`
- `minorVersion` = highest value of the minor version that this implementation can support for the required value of `releaseCode` and the returned value of `majorVersion`

If the preceding condition cannot be met, `SA_AIS_ERR_VERSION` is returned, and the version to which the `version` parameter points is set to:

if (implementation supports the required `releaseCode`)

`releaseCode` = required `releaseCode`

else {

if (implementation supports `releaseCode` higher than the required `releaseCode`)

`releaseCode` = the lowest value of the supported release codes that is higher than the required `releaseCode`

else

`releaseCode` = the highest value of the supported release codes that is lower than the required `releaseCode`

}

`majorVersion` = highest value of the major versions that this implementation can support for the returned `releaseCode`

`minorVersion` = highest value of the minor versions that this implementation can support for the returned values of `releaseCode` and `majorVersion`

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or a process that is providing the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_VERSION - The version provided in the structure to which the `version` parameter points is not compatible with the version of the Availability Management Framework implementation.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node because it is not a member node.

See Also

`saAmfSelectionObjectGet()`, `saAmfDispatch()`, `saAmfFinalize()`

7.5.2 saAmfSelectionObjectGet()

Prototype

```
SaAisErrorT saAmfSelectionObjectGet(
      SaAmfHandleT amfHandle,
      SaSelectionObjectT *selectionObject
);
```

Parameters

`amfHandle` - [in] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`selectionObject` - [out] A pointer to the operating system handle that the process can use to detect pending callbacks. The `SaSelectionObjectT` type is defined in [2].

Description

This function returns the operating system handle associated with the handle `amfHandle`. The invoking process can use the operating system handle to detect pending callbacks, instead of repeatedly invoking the `saAmfDispatch()` function for this purpose.

In a POSIX environment, the operating system handle is a file descriptor that is used with the `poll()` or `select()` system calls to detect incoming callbacks.

The operating system handle returned by `saAmfSelectionObjectGet()` is valid until `saAmfFinalize()` is invoked on the same handle `amfHandle`.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle amfHandle is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle amfHandle was acquired before the cluster node left the cluster membership.

See Also

saAmfInitialize_4(), saAmfDispatch()

7.5.3 saAmfDispatch()

Prototype

```
SaAisErrorT saAmfDispatch(
      SaAmfHandleT amfHandle,
      SaDispatchFlagsT dispatchFlags
);
```

Parameters

amfHandle - [in] The handle which was obtained by a previous invocation of the saAmfInitialize_4() function and which identifies this particular initialization of the Availability Management Framework. The SaAmfHandleT type is defined in Section 7.4.1 on page 245.

dispatchFlags - [in] Flags that specify the callback execution behavior of the saAmfDispatch() function, which have the values SA_DISPATCH_ONE, SA_DISPATCH_ALL, or SA_DISPATCH_BLOCKING. These flags are values of the SaDispatchFlagsT enumeration type, which is described in [2].

Description

In the context of the calling thread, this function invokes pending callbacks for the handle `amfHandle` in a way that is specified by the `dispatchFlags` parameter.

Return Values

`SA_AIS_OK` - The function completed successfully. This value is also returned if this function is being invoked with `dispatchFlags` set to `SA_DISPATCH_ALL` or `SA_DISPATCH_BLOCKING`, and the handle `amfHandle` has been finalized.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - The `dispatchFlags` parameter is invalid.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfInitialize_4()`, `saAmfSelectionObjectGet()`

7.5.4 saAmfFinalize()

Prototype

```
SaAisErrorT saAmfFinalize(
      SaAmfHandleT amfHandle
);
```

Parameters

`amfHandle` - [in] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

Description

The `saAmfFinalize()` function closes the association represented by the `amfHandle` parameter between the invoking process and the Availability Management Framework. The process must have invoked `saAmfInitialize_4()` before it invokes this function. A process must call this function once for each handle it acquired by invoking `saAmfInitialize_4()`.

If the `saAmfFinalize()` function completes successfully, it releases all resources acquired when `saAmfInitialize_4()` was called. Moreover, it unregisters all components registered for the particular handle. Furthermore, it stops any tracking associated with the particular handle and cancels all pending callbacks related to the particular handle.
Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

After `saAmfFinalize()` completes successfully, the handle `amfHandle` and the selection object associated with it are no longer valid.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

See Also

`saAmfInitialize_4()`

7.6 Component Registration

The functions in this section are used to register a component with the Availability Management Framework and to obtain the name of a component.

7.6.1 saAmfComponentRegister()

Prototype

```
SaAisErrorT saAmfComponentRegister(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      const SaNameT *proxyCompName
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The Availability Management Framework must maintain the list of components registered with each such handle. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component to be registered. The `SaNameT` type is defined in [2].

`proxyCompName` - [`in`] A pointer to the name of the proxy component that is registering the proxied component which is identified by the name to which `compName` points. The `proxyCompName` parameter is used only when a proxied component is being registered by a proxy component; otherwise, it must be set to NULL. The `SaNameT` type is defined in [2].

Description

This function registers the component identified by the name pointed to by the `compName` parameter with the Availability Management Framework. Registering a component informs the Availability Management Framework that the component is successfully instantiated and ready to take component service instance assignments.

If the registration succeeds, the invoking process becomes the registered process for the component identified by the name pointed to by the `compName` parameter (see Section 7.1.1).

A process of a proxy component may invoke this function to register one of its proxied components as a consequence of the Availability Management Framework having invoked a callback

- to request the proxy component either to instantiate a pre-instantiable proxied component or to assign a CSI to a non-pre-instantiable proxied component or
- to re-assign to the proxy component the active HA state for the proxy CSI for an already instantiated proxied component.

In these preceding cases, the registration of the proxied component must take place before the proxy component responds to the callback request by invoking the `saAmfResponse_4()` function.

To register itself, a contained component invokes this function from a process of its associated container component.

The registered process for a regular SA-aware component or for a contained component must have supplied in its `saAmfInitialize_4()` call the `saAmfCSISetCallback()`, `saAmfCSIRemoveCallback()`, and `saAmfComponentTerminateCallback()` callback functions.

The registered process for a container component must have supplied in its `saAmfInitialize_4()` call the `saAmfCSISetCallback()`, `saAmfCSIRemoveCallback()`, `saAmfComponentTerminateCallback()`, `saAmfContainedComponentInstantiateCallback()`, and `saAmfContainedComponentCleanupCallback()` callback functions.

Depending on the category of the proxied component it is configured to proxy, the registered process for a proxy component may need to supply in its `saAmfInitialize_4()` call additional callback functions to those mandatory for a regular SA-aware component. If the proxy component is configured to proxy a pre-instantiable proxied component, the registered process for the proxy component must supply the `saAmfProxiedComponentInstantiateCallback()` callback function.

If a proxy component is configured to proxy an external proxied component, the process (of the proxy component) that registers the proxied component must have supplied in its `saAmfInitialize_4()` call the `saAmfProxiedComponentCleanupCallback()` callback function in addition to those callbacks mandatory for a regular SA-aware component.

A component (SA-aware or proxied) must not register or (be registered) twice before having been unregistered, even if a different handle obtained by another invocation of the `saAmfInitialize_4()` call is used.

If an SA-aware component fails, it is implicitly unregistered by the Availability Management Framework. The same is true for a proxied component if its proxy fails, but the proxied component itself does not fail. If the proxied component fails, it is the task of its proxy to report an error on the failed component.

Return Values

SA_AIS_OK - The function returned successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INIT - The previous invocation of `saAmfInitialize_4()` to initialize the Availability Management Framework was incomplete, as one or more of the callback functions that are listed next were not supplied:

- If a regular SA-aware or contained component registers itself: `saAmfComponentTerminateCallback()`, `saAmfCSISetCallback()`, and `saAmfCSIRemoveCallback()`.
- If a container component registers itself: `saAmfComponentTerminateCallback()`, `saAmfCSISetCallback()`, `saAmfCSIRemoveCallback()`, `saAmfContainedComponentInstantiateCallback()`, and `saAmfContainedComponentCleanupCallback()`.
- If a proxy component registers itself: `saAmfComponentTerminateCallback()`, `saAmfCSISetCallback()`, `saAmfCSIRemoveCallback()`, and if the proxy is configured to proxy a pre-instantiable proxied component: `saAmfProxiedComponentInstantiateCallback()`.
- If a proxy component registers an external proxied component: `saAmfProxiedComponentCleanupCallback()`.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly. In particular, this value is returned if the value pointed to by `compName` is not the name of a configured component, or the names pointed to by `compName` or `proxyCompName` are not valid component DNs.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - The proxy component identified by the name to which proxyCompName refers has not been registered previously.

SA_AIS_ERR_EXIST - The component identified by the name to which compName refers has been registered previously with either the amfHandle handle or another handle obtained by a previous invocation of the saAmfInitialize_4() call.

SA_AIS_ERR_BAD_OPERATION - The proxy component which is identified by the name referred to by proxyCompName and which is registering a proxied component has not been assigned the proxy CSI with the active HA state through which the proxied component being registered is supposed to be proxied.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle amfHandle was acquired before the cluster node left the cluster membership.

See Also

SaAmfCSISetCallbackT, SaAmfCSIRemoveCallbackT, SaAmfComponentTerminateCallbackT, SaAmfProxiedComponentInstantiateCallbackT, SaAmfProxiedComponentCleanupCallbackT, SaAmfContainedComponentInstantiateCallbackT, SaAmfContainedComponentCleanupCallbackT, saAmfInitialize_4()

7.6.2 saAmfComponentNameGet()

Prototype

```
SaAisErrorT saAmfComponentNameGet(
      SaAmfHandleT amfHandle,
      SaNameT *compName
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`out`] A pointer to the name of the component to which the invoking process belongs. The `SaNameT` type is defined in [2].

Description

This function returns the name of the component to which the invoking process belongs. This function can be invoked by the process before its component has been registered with the Availability Management Framework by calling `saAmfComponentRegister()`. The component name provided by `saAmfComponentNameGet()` should be used by a process when it registers its local component.

As the Availability Management Framework does not control the creation of all processes that constitute a component, some conventions must be respected by the creators of these processes to allow the `saAmfComponentNameGet()` function to work properly in the different processes that constitute a component.

On operating systems supporting the concept of environment variables, the Availability Management Framework ensures that the `SA_AMF_COMPONENT_NAME` environment variable is properly set when it runs the `INSTANTIATE` command to create a component. It is the responsibility of the `INSTANTIATE` command, and more generally of any entity that creates processes for a component (also when the components are not instantiated by the Availability Management Framework), to ensure that the `SA_AMF_COMPONENT_NAME` environment variable is properly set to contain the component name when creating new processes. For more information about the environment variables supported by the Availability Management Framework, refer to Section 4.3 on page 209.

Note: It is not guaranteed that `saAmfComponentNameGet()` works for contained components. If it is not supported by the Availability Management Framework implementation, `SA_AIS_ERR_NOT_SUPPORTED` is returned.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - The system is out of required resources (other than memory).

`SA_AIS_ERR_NOT_EXIST` - The Availability Management Framework is not aware of any component associated with the invoking process.

`SA_AIS_ERR_NOT_SUPPORTED` - The Availability Management Framework returns this value if the `saAmfComponentNameGet()` function is not supported for contained components.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfComponentRegister()`, `saAmfInitialize_4()`

7.7 Passive Monitoring of Processes of a Component

This section describes the API functions that enable components to request the Availability Management Framework to perform passive monitoring of their processes.

7.7.1 saAmfPmStart_3()

Prototype

```
SaAisErrorT saAmfPmStart_3(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      SaInt64T processId,
      SaInt32T descendentsTreeDepth,
      SaAmfPmErrorsT pmErrors,
      SaAmfRecommendedRecoveryT recommendedRecovery
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component to which the monitored processes belong. The `SaNameT` type is defined in [2].

`processId` - [`in`] Identifier of a process to be monitored. The `SaInt64T` type is defined in [2].

`descendentsTreeDepth` - [`in`] Depth of the tree of descendents of the process identified by `processId` and that are also to be monitored. This parameter is of the `SaInt32T` type (defined in [2]) and can have the following values:

- A value of 0 indicates that no descendents of the designated process will be monitored.
- A value of 1 indicates that direct children of the designated process will be monitored.
- A value of 2 indicates that direct children and grand children of the designated process will be monitored, and so on.

- A value of –1 indicates that descendents at any level in the descendents tree will be monitored.

`pmErrors` - [`in`] Specifies the type of process errors to monitor. Monitoring for several errors can be requested in a single call by "ORing" different `SaAmfPmErrorsT` values (this type is defined in Section 7.4.2.1 on page 246):

- `SA_AMF_PM_NON_ZERO_EXIT` requests the monitoring of processes exiting with a nonzero exit status.
- `SA_AMF_PM_ZERO_EXIT` requests the monitoring of processes exiting with a zero exit status.

`recommendedRecovery` - [`in`] Recommended recovery to be performed by the Availability Management Framework. For details, refer to Section 7.4.7 on page 257 on the `SaAmfRecommendedRecoveryT` type.

Description

The `saAmfPmStart_3()` function requests the Availability Management Framework to start passive monitoring of specific errors that may occur to a process and to its descendents. Currently, only death of processes can be monitored. If one of the errors being monitored occurs for the process or for one of its descendents, the Availability Management Framework will automatically report an error on the component identified by the name to which `compName` refers (for details regarding error reports, see `saAmfComponentErrorReport_4()`). The recommended recovery action will be set according to the `recommendedRecovery` parameter.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - Either one or both of the cases that follow apply:

- The component identified by the name to which compName refers is not configured in the Availability Management Framework to execute on the local node.
- The process identified by processId does not exist on the local node.

SA_AIS_ERR_ACCESS - The Availability Management rejects the requested recommended recovery.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle amfHandle was acquired before the cluster node left the cluster membership.

See Also

saAmfPmStop(), saAmfComponentErrorReport_4(), saAmfInitialize_4()

7.7.2 saAmfPmStop()

Prototype

```
SaAisErrorT saAmfPmStop(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      SaAmfPmStopQualifierT stopQualifier,
      SaInt64T processId,
      SaAmfPmErrorsT pmErrors
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component to which the monitored processes belong. The `SaNameT` type is defined in [2].

`stopQualifier` - [`in`] Qualifies which processes should stop being monitored. This parameter is of the `SaAmfPmStopQualifierT` type (defined in Section 7.4.2.2 on page 246) and can have the following values:

- `SA_AMF_PM_PROC`: the Availability Management Framework stops monitoring the process identified by `processId`.
- `SA_AMF_PM_PROC_AND_DESCENDENTS`: the Availability Management Framework stops monitoring the process identified by `processId` and all its descendents.
- `SA_AMF_PM_ALL_PROCESSES`: the Availability Management Framework stops monitoring all processes that belong to the component identified by the name to which `compName` refers.

`processId` - [`in`] Identifier of the process for which passive monitoring is to be stopped. The `SaInt64T` type is defined in [2].

`pmErrors` - [`in`] Specifies the type of process errors that the Availability Management Framework should stop monitoring for the designated processes. Stopping the monitoring for several errors can be requested in a single call by “ORing” different `SaAmfPmErrorsT` values (this type is defined in Section 7.4.2.1 on page 246).

Description

The `saAmfPmStop()` function requests the Availability Management Framework to stop passive monitoring of specific errors that may occur to a set of processes belonging to a component.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - Either one, two, or all cases that follow apply:

- The component identified by the name to which `compName` refers is not configured in the Availability Management Framework to execute on the local node.
- The process identified by `processId` does not execute on the local node.
- The process identified by `processId` was not monitored by the Availability Management Framework for errors specified by `pmErrors`.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfInitialize_4()`, `saAmfPmStart_3()`

7.8 Component Health Monitoring

The following calls are used to monitor the health of a component.

7.8.1 saAmfHealthcheckStart()

Prototype

```
SaAisErrorT saAmfHealthcheckStart(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      const SaAmfHealthcheckKeyT *healthcheckKey,
      SaAmfHealthcheckInvocationT invocationType,
      SaAmfRecommendedRecoveryT recommendedRecovery
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component to be healthchecked. The `SaNameT` type is defined in [2].

`healthcheckKey` - [`in`] A pointer to the key of the healthcheck to be executed. Based on this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters. The `SaAmfHealthcheckKeyT` type is defined in Section 7.4.3.2 on page 247.

`invocationType` - [`in`] This parameter indicates whether the Availability Management Framework or the process itself will invoke the healthcheck calls. The `SaAmfHealthcheckInvocationT` type is defined in Section 7.4.3.1 on page 246.

`recommendedRecovery` - [`in`] Recommended recovery to be performed by the Availability Management Framework if the component fails a healthcheck. For details, refer to Section 7.4.7 on page 257 where the `SaAmfRecommendedRecoveryT` type is defined.

Description

This function starts healthchecks for the component designated by the name pointed to by `compName`. The variant of the healthcheck (component-invoked or framework-invoked) is specified by `invocationType`. If `invocationType` is `SA_HEALTHCHECK_AMF_INVOKED`, the `saAmfHealthcheckCallback()` callback function must have been supplied when the process invoked the `saAmfInitialize_4()` call.

If the component identified by the name to which the `compName` parameter points is a proxied component, the Availability Management Frameworks assumes that the invoking process belongs to its proxy component, that is, to the component that has the proxied's proxy CSI assigned active; otherwise, the Availability Management Frameworks assumes that the invoking process belongs to the component identified by the name to which the `compName` parameter points.

If a component wants to start more than one healthcheck, it should invoke this function once for each individual healthcheck. It is, however, not possible to have at a given time and on the same `amfHandle` two healthchecks started for the same component name and healthcheck key.

If the variant of the healthcheck is component-invoked, and the invoking process does not call the `saAmfHealthcheckConfirm()` function within the configured time interval for the healthcheck, and the `compName` parameter does not point to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter. Otherwise, if the `compName` parameter points to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for its proxy component.
The Availability Management Framework determines the configured time interval for the healthcheck referred to by the `healthcheckKey` parameter from one of two possible sources.

- If the healthcheck is specifically configured for the component referred to by the `compName` parameter, the Availability Management Framework obtains the time interval from the `saAmfHealthcheckMaxDuration` configuration attribute of the `SaAmfHealthcheck` configuration object class (see Section 8.14).
- If the healthcheck is not configured for the component referred to by the `compName` parameter, the Availability Management Framework obtains the time interval from the `saAmfHctDefMaxDuration` configuration attribute of the `SaAmfHealthcheckType` configuration object class instance associated with the component type object of the component (see Section 8.14).

A healthcheck is automatically stopped by the Availability Management Framework if

- the handle `amfHandle` is finalized, or
- the component identified by the name to which the `compName` parameter points is a proxied component, and its proxy CSI is no longer assigned active to the component that started the healthcheck.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INIT` - The previous invocation of `saAmfInitialize_4()` to initialize the Availability Management Framework was incomplete, as the `saAmfHealthcheckCallback()` callback function is missing, and `invocationType` specifies `SA_HEALTHCHECK_AMF_INVOKED`.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - The system is out of required resources (other than memory).

`SA_AIS_ERR_NOT_EXIST` - This value is returned if one or more of the cases that follow apply:

- The Availability Management Framework is not aware of a component designated by the name to which `compName` refers.
- The healthcheck identified by the key to which `healthcheckKey` points is not configured for the component designated by the name to which `compName` refers, or it is not configured for the component type associated with the component designated by the name to which `compName` refers.
- The component identified by the name to which the `compName` parameter points is a proxied component, but its proxy CSI is not assigned active to any proxy component.

SA_AIS_ERR_ACCESS - The Availability Management rejects the requested recommended recovery.

SA_AIS_ERR_EXIST - The healthcheck with the handle `amfHandle` has already been started for the component designated by the name to which `compName` refers and for the same value of the key to which `healthcheckKey` points.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaAmfHealthcheckCallbackT`, `saAmfHealthcheckConfirm()`, `saAmfHealthcheckStop()`, `saAmfInitialize_4()`

7.8.2 SaAmfHealthcheckCallbackT

Prototype

```
typedef void (*SaAmfHealthcheckCallbackT)(
     SaInvocationT invocation,
     const SaNameT *compName,
     SaAmfHealthcheckKeyT *healthcheckKey
);
```

Parameters

`invocation` - [`in`] This parameter identifies a particular invocation of the callback function. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` function. The `SaInvocationT` type is defined in [2].

`compName` - [`in`] A pointer to the name of the component that must undergo the particular healthcheck. The `SaNameT` type is defined in [2].

`healthcheckKey` - [`in`] A pointer to the key of the healthcheck to be executed. The `SaAmfHealthcheckKeyT` type is defined in Section 7.4.3.2 on page 247.

Description

The Availability Management Framework invokes this callback to request the invoked process to perform a healthcheck specified by the key pointed to by the `healthcheckKey` parameter for the component identified by the name pointed to by the `compName` parameter.

This callback is invoked in the context of a thread of the process that started the healthcheck operation by invoking the `saAmfHealthcheckStart()` function, when this thread invokes the `saAmfDispatch()` function with the handle `amfHandle` that was specified when the healthcheck operation was started.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework about the completion of the healthcheck by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- `SA_AIS_OK` - The healthcheck completed successfully.
- `SA_AIS_ERR_FAILED_OPERATION` - The component failed to successfully execute the given healthcheck and has reported an error on the faulty component by invoking `saAmfComponentErrorReport_4()`.

Any other error code set in the `error` parameter in the response will be treated by the Availability Management Framework as if the caller had set the `error` parameter to `SA_AIS_ERR_FAILED_OPERATION`.

If the invoked process responds by calling `saAmfResponse_4()` with the `error` parameter set to `SA_AIS_ERR_FAILED_OPERATION` within the time interval configured for the healthcheck, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter.
The Availability Management Framework determines the configured time interval for the healthcheck referred to by the `healthcheckKey` parameter from one of two possible sources.

- If the healthcheck is specifically configured for the component referred to by the `compName` parameter, the Availability Management Framework obtains the time interval from the `saAmfHealthcheckMaxDuration` configuration attribute of the `SaAmfHealthcheck` configuration object class (see Section 8.14).
- If the healthcheck is not configured for the component referred to by the `compName` parameter, the Availability Management Framework obtains the time interval from the `saAmfHctDefMaxDuration` configuration attribute of the `SaAmfHealthcheckType` configuration object class instance associated with the component type object of the component (see Section 8.14).

If the invoked process does not respond by calling `saAmfResponse_4()` within the aforementioned time interval, and the `compName` parameter does not point to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter; otherwise, if the `compName` parameter points to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs).

See Also

`saAmfResponse_4()`, `saAmfHealthcheckStart()`, `saAmfComponentErrorReport_4()`, `saAmfDispatch()`

7.8.3 saAmfHealthcheckConfirm()

Prototype

```
SaAisErrorT saAmfHealthcheckConfirm(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      const SaAmfHealthcheckKeyT *healthcheckKey,
      SaAisErrorT healthcheckResult
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component for which the healthcheck result is being reported. The `SaNameT` type is defined in [2].

`healthcheckKey` - [`in`] A pointer to the key of the healthcheck whose result is being reported. Based on this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters. The `SaAmfHealthcheckKeyT` type is defined in Section 7.4.3.2 on page 247.

`healthcheckResult` - [in] This parameter of `SaAisErrorT` type (defined in [2]) indicates the result of the healthcheck performed by the component. This parameter can take one of the following values:

- `SA_AIS_OK` - The healthcheck completed successfully.
- `SA_AIS_ERR_FAILED_OPERATION`: the component failed to successfully execute the given healthcheck and has reported an error on itself by invoking `saAmfComponentErrorReport_4()`.

Any other error code set in the `healthcheckResult` parameter will be treated by the Availability Management Framework as if the caller had set the `healthcheckResult` parameter to `SA_AIS_ERR_FAILED_OPERATION`.

Description

This function allows a process to inform the Availability Management Framework that it has performed the healthcheck identified by the key pointed to by `healthcheckKey` for the component designated by the name to which `compName` points, and whether the healthcheck was successful or not.

The invoking process must be the same process that started the healthcheck by invoking the `saAmfHealthcheckStart()` function.

If the invoking process sets the `healthcheckResult` parameter to `SA_AIS_ERR_FAILED_OPERATION` within the time interval configured for the healthcheck, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for this component.
The Availability Management Framework determines the configured time interval for the healthcheck referred to by the `healthcheckKey` parameter from one of two possible sources.

- If the healthcheck is specifically configured for the component referred to by the `compName` parameter, the Availability Management Framework obtains the time interval from the `saAmfHealthcheckMaxDuration` configuration attribute of the `SaAmfHealthcheck` configuration object class (see Section 8.14).
- If the healthcheck is not configured for the component referred to by the `compName` parameter, the Availability Management Framework obtains the time interval from the `saAmfHctDefMaxDuration` configuration attribute of the `SaAmfHealthcheckType` configuration object class instance associated with the component type object of the component (see Section 8.14).

If the process that initiated a component-invoked healthcheck by calling the `saAmfHealthcheckStart()` function does not call the `saAmfHealthcheckConfirm()` function within the configured time interval for the healthcheck, and the `compName` parameter does not point to a proxied component,

the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter; otherwise, if the `compName` parameter points to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for its proxy component.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly. In particular, this value is returned if the calling process is not the process that started the healthcheck by invoking `saAmfHealthcheckStart()`.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - The system is out of required resources (other than memory).

`SA_AIS_ERR_NOT_EXIST` - Either one or both of the cases that follow apply:

- The Availability Management Framework is not aware of a component designated by the name to which `compName` points.
- No component-invoked healthcheck has been started for the component designated by the name to which `compName` points and for the key referred to by `healthcheckKey`.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfHealthcheckStart()`, `saAmfComponentErrorReport_4()`, `saAmfInitialize_4()`

7.8.4 saAmfHealthcheckStop()

Prototype

```
SaAisErrorT saAmfHealthcheckStop(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      const SaAmfHealthcheckKeyT *healthcheckKey
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component for which healthchecks are to be stopped. The `SaNameT` type is defined in [2].

`healthcheckKey` - [`in`] A pointer to the key of the healthcheck to be stopped. Based on this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters. The `SaAmfHealthcheckKeyT` type is defined in Section 7.4.3.2 on page 247.

Description

This function is used to stop the healthcheck referred to by the key pointed by `healthcheckKey` for the component designated by the name to which `compName` points.

Return Values

SA_AIS_OK - The function returned successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly. A specific example is when the calling process is not the process that has started the associated healthcheck.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - Either one or both of the cases that follow apply:

- The Availability Management Framework is not aware of a component designated by the name to which `compName` points.
- No healthcheck has been started for the component designated by the name to which `compName` points and for the key to which `healthcheckKey` refers.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfHealthcheckStart()`, `saAmfInitialize_4()`

7.9 Component Service Instance Management

The following calls are used to manage the HA state of components on behalf of the component service instances that they support.

7.9.1 saAmfHAStateGet()

Prototype

```
SaAisErrorT saAmfHAStateGet(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      const SaNameT *csiName,
      SaAmfHAStateT *haState
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the component for which the information is requested. The `SaNameT` type is defined in [2].

`csiName` - [`in`] A pointer to the name of the component service instance for which the information is requested. The `SaNameT` type is defined in [2].

`haState` - [`out`] A pointer to the HA state that the Availability Management Framework has currently assigned to the component identified by the name to which `compName` points for the component service instance identified by the name to which `csiName` refers. The HA state is active, standby, quiescing, or quiesced, as defined by the `SaAmfHAStateT` enumeration type (see Section 7.4.4.1 on page 247).

Description

The Availability Management Framework returns the HA state of a component identified by the name to which `compName` refers for the component service instance identified by the name to which `csiName` refers.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - The system is out of required resources (other than memory).

`SA_AIS_ERR_NOT_EXIST` - The component identified by the name to which `compName` points has not been registered with the Availability Management Framework.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaAmfCSISetCallbackT`, `saAmfInitialize_4()`

7.9.2 SaAmfCSISetCallbackT

Prototype

```
typedef void (*SaAmfCSISetCallbackT)(
     SaInvocationT invocation,
     const SaNameT *compName,
     SaAmfHAStateT haState,
     SaAmfCSIDescriptorT csiDescriptor
);
```

Parameters

`invocation` - [in] This parameter identifies a particular invocation of the callback function. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` or `saAmfCSIQuiescingComplete()` functions. The `SaInvocationT` type is defined in [2].

`compName` - [in] A pointer to the name of the component to which a new component service instance is assigned or for which the HA state of one or all supported component service instances is changed. The `SaNameT` type is defined in [2].

`haState` - [in] The new HA state to be assumed by the component identified by the name to which `compName` points for the component service instance identified by `csiDescriptor`, or for all component service instances already supported by the component (if `SA_AMF_CSI_TARGET_ALL` is set in `csiFlags` of the `csiDescriptor` parameter). The `SaAmfHAStateT` type is defined in Section 7.4.4.1 on page 247.

`csiDescriptor` - [in] The descriptor with information about the component service instances targeted by this callback invocation. The `SaAmfCSIDescriptorT` type is defined in Section 7.4.5.5 on page 254.

Description

The Availability Management Framework invokes this callback to request that the component identified by the name to which `compName` points assume the HA state specified by `haState` for one or all component service instances.

The component service instances targeted by this call along with additional information about them are provided by the `csiDescriptor` parameter.

If the haState parameter indicates that the new HA state for the CSIs is SA_AMF_HA_QUIESCING, the invoked process must notify the Availability Management Framework that the assignment has been received by invoking the saAmfResponse_4() function. Subsequently, when the CSIs have been quiesced, the invoked process must notify the Availability Management Framework by invoking the saAmfCSIQuiescingComplete() function. When the process invokes both the saAmfResponse_4() and saAmfCSIQuiescingComplete() functions, the process provides invocation as an in parameter.

If the compName parameter refers to a non-pre-instantiable proxied component, the Availability Management Framework invokes this callback in the context of a thread of the registered process for its proxy component when this thread calls saAmfDispatch() with the handle amfHandle that was specified when the proxy component was registered by invoking saAmfComponentRegister(). For all other categories of components referred to by the compName parameter, this callback is invoked in the context of a thread of a registered process when this thread calls saAmfDispatch() with the handle amfHandle that was specified when the component identified by the name referred to by compName was registered by invoking saAmfComponentRegister().

If the compName parameter refers to a non-pre-instantiable proxied component, the invoked process (which must be the registered process for the proxy component) must register the proxied component before the invoked process responds to this callback request by invoking the saAmfResponse_4() function.

The Availability Management Framework sets invocation, and the process returns invocation as an in parameter when it responds to the Availability Management Framework by invoking the saAmfResponse_4() function. The error parameter in the invocation of the saAmfResponse_4() function should be set to one of the following values:

- SA_AIS_OK - The component executed the saAmfCSISetCallback() function successfully.
- SA_AIS_ERR_NOT_READY - The component has changed its HA readiness state for the given component service instance to indicate that it cannot assume the HA state specified by haState for at least one component service instance.
- SA_AIS_ERR_FAILED_OPERATION - The component failed to assume the HA state specified by haState for at least one component service instance.

Any other error code set in the error parameter in the response will be treated by the Availability Management Framework as if the caller had set the error parameter to SA_AIS_ERR_FAILED_OPERATION.

If the invoked process responds by calling `saAmfResponse_4()` with the `error` parameter set to `SA_AIS_ERR_FAILED_OPERATION` within the time interval configured by the value of the `saAmfCompCSISetCallbackTimeout` configuration attribute of the `SaAmfComp` configuration object class (see Section 8.13.2) for the component referred to by the `compName` parameter, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter.

However, a response with the error parameter set to `SA_AIS_ERR_NOT_READY`, does not trigger a recovery action from the Availability Management Framework if the invoked process first calls the `saAmfHAReadinessStateSet()` function to set the HA readiness state of the component for the assigned component service instance to a value indicating that the component is not ready for the requested assignment, and it then invokes the `saAmfResponse_4()` function to respond to this callback. These values of the HA readiness state are:

- `SA_AMF_HARS_NOT_READY_FOR_ACTIVE` or `SA_AMF_HARS_NOT_READY_FOR_ASSIGNMENT` for an active or quiescing assignment, or
- `SA_AMF_HARS_NOT_READY_FOR_ASSIGNMENT` for a standby assignment.

Only these combinations of HA state assignments and HA readiness state values are considered valid reasons to reject an assignment without triggering a recovery action. This means in particular that

- a component must always be ready to accept the assignment of a component service instance in any HA state if the HA readiness state of the component for that component service instance is either `SA_AMF_HARS_READY_FOR_ASSIGNMENT` or `SA_AMF_HARS_READY_FOR_ACTIVE_DEGRADED` and that
- a component must always be ready to accept the quiesced assignment of a component service instance irrespective of the HA readiness state of the component for that component service instance.

If the invoked process does not respond by calling `saAmfResponse_4()` within the aforementioned configured time interval, and the `compName` parameter does not point to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter. Otherwise, if the `compName` parameter points to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs).

If the haState parameter specifies SA_AMF_HA_QUIESCING, and the invoked process responds by calling the saAmfResponse_4() function with the error parameter set to SA_AIS_OK within the time period configured by the value of the saAmfCompCSISetCallbackTimeout configuration attribute of the SaAmfComp configuration object class (see Section 8.13.2) for the component referred to by the compName parameter, but the invoked process fails to subsequently report that this component has successfully quiesced (by invoking the saAmfCSIQuiescingComplete() function) within the configured saAmfCompQuiescingCompleteTimeout time period for the same component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs). The invocation of the saAmfCSIQuiescingComplete() function is not required if—within the saAmfCompQuiescingCompleteTimeout time period—the Availability Management Framework sets a new HA state for all the CSIs originally targeted by this call.

See Also

saAmfResponse_4(), saAmfCSIQuiescingComplete(), saAmfComponentRegister(), saAmfDispatch()

7.9.3 SaAmfCSIRemoveCallbackT

Prototype

```
typedef void (*SaAmfCSIRemoveCallbackT)(
      SaInvocationT invocation,
      const SaNameT *compName,
      const SaNameT *csiName,
      SaAmfCSIFlagsT csiFlags
);
```

Parameters

invocation - [in] This parameter identifies a particular invocation of the callback function. The invoked process returns invocation when it responds to the Availability Management Framework by calling the saAmfResponse_4() function. The SaInvocationT type is defined in [2].

compName - [in] A pointer to the name of the component from which all component service instances or the component service instance identified by the name referred to by csiName will be removed. The SaNameT type is defined in [2].

`csiName` - [`in`] A pointer to the name of the component service instance that must be removed from the component identified by the name to which `compName` points. The `SaNameT` type is defined in [2].

`csiFlags` - [`in`] This flag specifies whether one or more component service instances are affected. It can contain one of the values `SA_AMF_TARGET_ONE` or `SA_AMF_TARGET_ALL`. The `SaAmfCSIFlagsT` type is defined in Section 7.4.5.1 on page 250.

Description

The Availability Management Framework requests the invoked process to remove from the component identified by the name referred to by `compName` one or all component service instances from the set of component service instances being supported.

If the value of `csiFlags` is `SA_AMF_TARGET_ONE`, `csiName` points to the name of the component service instance that must be removed. If the value of `csiFlags` is `SA_AMF_TARGET_ALL`, `csiName` is NULL, and the component must remove all component service instances.

The Availability Management Framework invokes this callback in the context of a thread of the registered process for the component identified by the name referred to by `compName` when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the component was registered by invoking `saAmfComponentRegister()`.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- `SA_AIS_OK` - The component executed the `saAmfCSIRemoveCallback()` function successfully.
- `SA_AIS_ERR_FAILED_OPERATION` - The component failed to remove at least one component service instance.

Any other error code set in the `error` parameter in the response will be treated by the Availability Management Framework as if the caller had set the `error` parameter to `SA_AIS_ERR_FAILED_OPERATION`.

If the invoked process responds by calling `saAmfResponse_4()` with the `error` parameter set to `SA_AIS_ERR_FAILED_OPERATION` within the time interval configured by the value of the `saAmfCompCSIRmvCallbackTimeout` configuration attribute of the `SaAmfComp` configuration object class (see Section 8.13.2) for the

component referred to by the compName parameter, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the compName parameter.
If the invoked process does not respond by calling saAmfResponse_4() within the aforementioned configured time interval, and the compName parameter does not point to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the compName parameter. Otherwise, if the compName parameter points to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs).

See Also

saAmfResponse_4(), saAmfComponentRegister(), saAmfDispatch()

7.9.4 saAmfCSIQuiescingComplete()

Prototype

```
SaAisErrorT saAmfCSIQuiescingComplete(
      SaAmfHandleT amfHandle,
      SaInvocationT invocation,
      SaAisErrorT error
);
```

Parameters

amfHandle - [in] The handle which was obtained by a previous invocation of the saAmfInitialize_4() function and which identifies this particular initialization of the Availability Management Framework. The SaAmfHandleT type is defined in Section 7.4.1 on page 245.

invocation - [in] The invocation parameter that the Availability Management Framework assigned when it invoked the saAmfCSISetCallback() callback function to request the component referred to by the compName parameter in the corresponding invocation of the saAmfCSISetCallback() callback function to enter the SA_AMF_HA_QUIESCING HA state for a particular component service instance or for all component service instances assigned to it. The SaInvocationT type is defined in [2].

`error` - [`in`] The component returns the status of the completion of the quiescing operation in this parameter (of `SaAisErrorT` type, defined in [2]), which has one of the following values:

- `SA_AIS_OK` - The component referred to by the `compName` parameter in the corresponding invocation of the `saAmfCSISetCallback()` callback function stopped successfully its activity related to a particular component service instance or to all component service instances assigned to it.
- `SA_AIS_ERR_FAILED_OPERATION` - The component referred to by the `compName` parameter in the corresponding invocation of the `saAmfCSISetCallback()` callback function failed to stop its activity related to a particular component service instance or to any of a set of component service instances assigned to it. Some of the actions required during quiescing might not have been performed.

If any other error code is returned in this parameter, it will be treated by the Availability Management Framework as if the caller had returned `SA_AIS_ERR_FAILED_OPERATION`.

Description

A process invokes this call to notify the Availability Management Framework whether the component referred to by the `compName` parameter in the corresponding invocation of the `saAmfCSISetCallback()` callback function, which has assigned the `SA_AMF_HA_QUIESCING` state to this component, has successfully stopped its activity related to a particular component service instance or to all component service instances assigned to it. The `invocation` parameter associates this call of `saAmfCSIQuiescingComplete()` with the corresponding invocation of the `saAmfCSISetCallback()` callback function.

It is possible that the component is unable to successfully complete the ongoing work due to, for example, a failure in the component. If possible, the component or its proxy (if it is a proxied component) should notify the Availability Management Framework of this fact by invoking this function. The error parameter specifies whether the component has stopped cleanly as requested.

If the error parameter is set to `SA_AIS_ERR_FAILED_OPERATION`, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the component referred to by the `compName` parameter in the corresponding `saAmfCSISetCallback()` callback function.

This function may only be called by the registered process for a component. The `amfHandle` must be the same that was used to register the component identified by the name referred to by the `compName` parameter in the corresponding invocation of the `saAmfCSISetCallback()` callback function.

Return Values

SA_AIS_OK - The function returned successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is set incorrectly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory, and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - The `invocation` parameter does not identify an invocation of the `saAmfCSISetCallback()` callback function for which the call of `saAmfCSIQuiescingComplete()` is outstanding.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaAmfCSISetCallbackT`, `saAmfResponse_4()`, `saAmfComponentRegister()`, `saAmfInitialize_4()`

7.9.5 saAmfHAReadinessStateSet()

```
SaAisErrorT saAmfHAReadinessStateSet(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      const SaNameT *csiName,
      SaAmfHAReadinessStateT haReadinessState,
      SaNtfCorrelationIdsT *correlationIds
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [`in`] A pointer to the name of the pre-instantiable component. The `SaNameT` type is defined in [2].

`csiName` - [`in`] A pointer to the name of the component service instance. If `csiName` is equal to NULL, all component service instances that are assigned or can potentially be assigned to the component are considered. The `SaNameT` type is defined in [2].

`haReadinessState` - [`in`] The HA Readiness state that the Availability Management Framework must set for the component identified by the name to which `compName` points for the component service instance identified by the name to which `csiName` refers (or for any component service instance that is assigned or can potentially be assigned to the component, if `csiName` is NULL). The HA readiness state is ready-for-assignment, ready-for-active-degraded, not-ready-for-active, or not-ready-for-assignment, as defined by the `SaAmfHAReadinessStateT` enumeration type (see Section 7.4.4.7 on page 249).

`correlationsIds` – [`in/out`] Pointer to correlation identifiers associated with the HA readiness state change. `rootCorrelationId` and `parentCorrelationId` are `in` parameters and hold the root and parent correlation identifiers, respectively. These correlation identifiers are included by the Availability Management Framework in its own notifications triggered by this change. The `rootCorrelationId` and `parentCorrelationId` may hold the same value. If both correlation identifiers are set to `SA_NTF_IDENTIFIER_UNUSED` (that is, there is no notification with which this

error report may be correlated), the Availability Management Framework returns in `notificationId` the identifier of the HA readiness state change notification it sends as a consequence of this call. The `SaNtfCorrelationIdsT` type is defined in [3].

Description

This function is invoked to set the HA readiness state of the pre-instantiable component identified by the name to which `compName` refers for the component service instance identified by the name to which `csiName` refers. If `csiName` is set to NULL, this function sets the HA readiness state of the component identified by the name to which `compName` refers for all component service instances that are assigned or can potentially be assigned to the component.

In case of success, the new value of the HA readiness state for the affected component service instances is equal to `haReadinessState`.

The change of the HA readiness state of a component for a component service instance may force the Availability Management Framework to change the current assignments for the component service instance. In particular:

- If the component has the active or quiescing assignment for the component service instance and its HA readiness state for that component service instance is set to SA_AMF_HARS_NOT_READY_FOR_ACTIVE, the Availability Management Framework must either remove the assignment from the component or change the assignment to standby.
- If the component is assigned any HA state for the component service instance, and the HA readiness state of the component for that component service instance is set to SA_AMF_HARS_NOT_READY_FOR_ASSIGNMENT, the Availability Management Framework must remove the assignment from the component.

When the Availability Management Framework invokes the `SaAmfCSISetCallbackT` callback function on the registered process for a component to request the component to take a particular component service instance assignment, the invoked process can call the `saAmfHAReadinessStateSet()` function to indicate that the component is not ready to take the assignment. The `saAmfHAReadinessStateSet()` function must be called before the invoked process responds to the callback invocation with the SA_AIS_ERR_NOT_READY error (by calling the `saAmfResponse_4()` function). The component can reject an active assignment by setting its HA readiness state for the component service instance to SA_AMF_HARS_NOT_READY_FOR_ACTIVE or SA_AMF_HARS_NOT_READY_FOR_ASSIGNMENT. The component can reject a standby assignment by setting its HA readiness state for the component service instance to SA_AMF_HARS_NOT_READY_FOR_ASSIGNMENT.

If a component is already assigned an HA state for a particular component service instance, and the component determines that it can no longer support that component service instance assignment or only in a limited capacity, the registered process for the component can call `saAmfHAReadinessStateSet()` to indicate that the component can no longer support the assignment or is limited in how it can support the assignment.

The `amfHandle` in the `saAmfHAReadinessStateSet()` call must be the same as the one used in the `saAmfComponentRegister()` call to register the component.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized, or the component referred to by the `compName` parameter has not been registered using this handle.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly. In particular:

- `compName` does not point to a valid component name, or `compName` does not identify a pre-instantiable component registered by the invoking process with the handle `amfHandle`.
- `csiName` does not point to the name of a component service instance that may be assigned to the component identified by the name to which `compName` points, or
- `haReadinessState` is not a valid HA readiness state.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - The system is out of required resources (other than memory).

`SA_AIS_ERR_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_NO_OP - The HA readiness state specified by the haReadinessState parameter for the component service instance(s) specified by the csiName parameter is the same as the existing HA readiness state for the identified component for the identified component service instance(s).

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle amfHandle was acquired before the cluster node left the cluster membership.

See Also

SaAmfCSISetCallbackT, saAmfResponse_4(), saAmfComponentRegister(), saAmfInitialize_4()

7.10 Component Life Cycle

This section describes the callback function to request a component to terminate. It contains also additional callback functions that proxy and container components export to enable the Availability Management Framework to manage proxied and contained components.

7.10.1 SaAmfComponentTerminateCallbackT

Prototype

```
typedef void (*SaAmfComponentTerminateCallbackT)(
          SaInvocationT invocation,
          const SaNameT *compName
);
```

Parameters

`invocation` - [in] This parameter identifies a particular invocation of this callback. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` function. The `SaInvocationT` type is defined in [2].

`compName` - [in] A pointer to the name of the component to be terminated. The `SaNameT` type is defined in [2].

Description

The Availability Management Framework requests the component identified by the name referred to by `compName` to terminate. To terminate a proxied component, the Availability Management Framework invokes this function on the proxy component that is “proxying” the component identified by the name to which `compName` points.

The component identified by the name referred to by `compName` is expected to release all acquired resources and to terminate itself. The invoked process responds by invoking the `saAmfResponse_4()` function.

This callback is invoked in the context of a thread of the registered process for the component identified by the name referred to by `compName` when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the component was registered by invoking `saAmfComponentRegister()`.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- `SA_AIS_OK` - The function completed successfully.
- `SA_AIS_ERR_FAILED_OPERATION` - The component identified by the name to which `compName` points failed to terminate.

Any other error code set in the `error` parameter in the response will be treated by the Availability Management Framework as if the caller had set the `error` parameter to `SA_AIS_ERR_FAILED_OPERATION`.

If the invoked process responds by calling `saAmfResponse_4()` with the `error` parameter set to `SA_AIS_ERR_FAILED_OPERATION` within the time interval configured by the value of the `saAmfCompTerminateCallbackTimeout` configuration attribute of the `SaAmfComp` configuration object class (see Section 8.13.2) for the component referred to by the `compName` parameter, the Availability Management Framework must execute the cleanup procedure for the component referred to by the `compName` parameter.
If the invoked process does not respond by calling `saAmfResponse_4()` within the aforementioned time interval, and the `compName` parameter does not point to a proxied component, the Availability Management Framework must execute the cleanup procedure for the component referred to by the `compName` parameter. Otherwise, if the `compName` parameter points to a proxied component, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs).

See Also

`saAmfResponse_4()`, `saAmfComponentRegister()`, `saAmfDispatch()`

7.10.2 SaAmfProxiedComponentInstantiateCallbackT

Prototype

```
typedef void (*SaAmfProxiedComponentInstantiateCallbackT)(
     SaInvocationT invocation,
     const SaNameT *proxiedCompName
);
```

Parameters

`invocation` - [`in`] This parameter identifies a particular invocation of this callback function. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` function. The `SaInvocationT` type is defined in [2].

`proxiedCompName` - [`in`] A pointer to the name of the proxied component to be instantiated. The `SaNameT` type is defined in [2].

Description

The Availability Management Framework requests a proxy component to instantiate a pre-instantiable proxied component identified by the name to which `proxiedCompName` points.

This callback is invoked in the context of a thread of the registered process for a proxy component when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the proxy component was registered by calling `saAmfComponentRegister()`.

The proxy component must register the proxied component referred to by the `proxiedCompName` parameter before the invoked process (which is the registered process for the proxy component) responds to this callback request by invoking the `saAmfResponse_4()` function.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- SA_AIS_OK - The function completed successfully.
- SA_AIS_ERR_FAILED_OPERATION - The proxy component failed to instantiate the proxied component. It is useless for the Availability Management Framework to attempt to instantiate the proxied component again.
- SA_AIS_ERR_TRY_AGAIN - The proxy component failed to instantiate the proxied component. The Availability Management Framework retries to instantiate the proxied component based on the configuration attributes described for the INSTANTIATE CLC-CLI command (see Section 4.6 on page 211).

Any other error code set in the error parameter in the response will be treated by the Availability Management Framework as if the caller had set the error parameter to SA_AIS_ERR_FAILED_OPERATION.

If the invoked process responds by calling saAmfResponse_4() with the error parameter set to SA_AIS_ERR_FAILED_OPERATION within the time interval configured by the value of the saAmfCompInstantiateTimeout configuration attribute of the SaAmfComp configuration object class (see Section 8.13.2) for the proxied component referred to by the proxiedCompName parameter, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for this proxied component.
If the invoked process does not respond by calling saAmfResponse_4() within the aforementioned time interval, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs).

See Also

saAmfResponse_4(), saAmfComponentRegister(), saAmfDispatch(), SaAmfProxiedComponentCleanupCallbackT

7.10.3 SaAmfProxiedComponentCleanupCallbackT

Prototype

```
typedef void (*SaAmfProxiedComponentCleanupCallbackT)(
     SaInvocationT invocation,
     const SaNameT *proxiedCompName
);
```

Parameters

`invocation` - [`in`] This parameter identifies a particular invocation of this callback function. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` function. The `SaInvocationT` type is defined in [2].

`proxiedCompName` - [`in`] A pointer to the name of the proxied component to be abruptly terminated. The `SaNameT` type is defined in [2].

Description

The Availability Management Framework requests a proxy component to abruptly terminate a proxied component identified by the name to which `proxiedCompName` points.

For a non-pre-instantiable proxied component, this callback is invoked in the context of a thread of the registered process for a proxy component when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the proxy component was registered by calling `saAmfComponentRegister()`.

For a pre-instantiable proxied component, this callback is invoked in the context of a thread of the registered process for the proxied component when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the proxied component was registered by calling `saAmfComponentRegister()`.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- SA_AIS_OK - The function completed successfully.
- SA_AIS_ERR_FAILED_OPERATION - The proxy component failed to abruptly terminate the proxied component. The Availability Management Framework might issue a further attempt to abruptly terminate the proxied component.

Any other error code set in the error parameter in the response will be treated by the Availability Management Framework as if the caller had set the error parameter to SA_AIS_ERR_FAILED_OPERATION.

If the invoked process responds by calling v with the error parameter set to SA_AIS_ERR_FAILED_OPERATION within the time interval configured by the value of the saAmfCompCleanupTimeout configuration attribute of the SaAmfComp configuration object class (see Section 8.13.2) for the proxied component referred to by the proxiedCompName parameter, the Availability Management Framework may issue a CLEANUP command (see Section 4.8) if it is available for this proxied component. For an external proxied component that had any active HA assignment for any CSI protected in a redundancy model that permits one such assignment, the CSI must stay unassigned until an administrative action is performed to terminate the failed component.
If the invoked process does not respond by calling saAmfResponse_4() within the aforementioned time interval, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the proxy component (that is, for the component to which the invoked process belongs).

See Also

saAmfResponse_4(), saAmfComponentRegister(), saAmfDispatch(), SaAmfProxiedComponentInstantiateCallbackT

7.10.4 SaAmfContainedComponentInstantiateCallbackT

Prototype

```
typedef void (*SaAmfContainedComponentInstantiateCallbackT)(
     SaInvocationT invocation,
     const SaNameT *containedCompName
);
```

Parameters

`invocation` - [`in`] This parameter identifies a particular invocation of this callback function. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` function. The `SaInvocationT` type is defined in [2].

`containedCompName` - [`in`] A pointer to the name of the contained component to be instantiated. The `SaNameT` type is defined in [2].

Description

The Availability Management Framework requests a container component to instantiate a contained component identified by the name to which `containedCompName` points. This callback is invoked by the Availability Management Framework only if the container component is assigned active for the container CSI that is configured to handle the life cycle of the contained component.

This callback is invoked in the context of a thread of the registered process for a container component when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the container component was registered by calling `saAmfComponentRegister()`.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- `SA_AIS_OK` - The function completed successfully. The container component becomes the associated container for the contained component.
  The successful completion of this function does not imply the successful instantiation of the contained component. The instantiation is considered successful only when the contained component registers itself with the Availability Management Framework.
- `SA_AIS_ERR_FAILED_OPERATION` - The container component failed to instantiate the contained component. It is useless for the Availability Management Framework to attempt to instantiate the contained component again.
- `SA_AIS_ERR_TRY_AGAIN` - The container component failed to instantiate the contained component. The Availability Management Framework retries to instantiate the contained component based on the configuration attributes described for the `INSTANTIATE` CLC-CLI command (see Section 4.6 on page 211).

Any other error code set in the `error` parameter in the response will be treated by the Availability Management Framework as if the caller had set the `error` parameter to `SA_AIS_ERR_FAILED_OPERATION`.

If the invoked process responds by calling `saAmfResponse_4()` with the `error` parameter set to `SA_AIS_ERR_FAILED_OPERATION` or the component referred to by the `containedCompName` parameter does not register within the time interval configured by the value of the `saAmfCompInstantiateTimeout` configuration attribute of the `SaAmfComp` configuration object class (see Section 8.13.2) for the component referred to by the `containedCompName` parameter, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for this contained component.
If the invoked process does not respond by calling `saAmfResponse_4()` within the aforementioned time interval, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the container component.

See Also

`saAmfResponse_4()`, `saAmfComponentRegister()`, `saAmfDispatch()`, `SaAmfContainedComponentCleanupCallbackT`

7.10.5 SaAmfContainedComponentCleanupCallbackT

Prototype

```
typedef void (*SaAmfContainedComponentCleanupCallbackT)(
      SaInvocationT invocation,
      const SaNameT *containedCompName
);
```

Parameters

`invocation` - [in] This parameter identifies a particular invocation of this callback function. The invoked process returns `invocation` when it responds to the Availability Management Framework by calling the `saAmfResponse_4()` function. The `SaInvocationT` type is defined in [2].

`containedCompName` - [in] A pointer to the name of the contained component to be abruptly terminated. The `SaNameT` type is defined in [2].

Description

The Availability Management Framework requests a container component to abruptly terminate a contained component identified by the name to which `containedCompName` points. This callback is invoked by the Availability Management Framework only if the container component is the associated container component for the contained component, that is, only if the container component has instantiated the contained component as a result of a successful invocation of the `saAmfContainedComponentInstantiateCallback()` callback on the registered process for the container component.

This callback is invoked in the context of a thread of the registered process for a container component when this thread calls `saAmfDispatch()` with the handle `amfHandle` that was specified when the container component was registered by calling `saAmfComponentRegister()`.

The Availability Management Framework sets `invocation`, and the component returns `invocation` as an `in` parameter when it responds to the Availability Management Framework by invoking the `saAmfResponse_4()` function. The `error` parameter in the invocation of the `saAmfResponse_4()` function should be set to one of the following values:

- `SA_AIS_OK` - The function completed successfully.
- `SA_AIS_ERR_FAILED_OPERATION` - The container component failed to abruptly terminate the contained component. The Availability Management Framework might issue a further attempt to abruptly terminate the contained component.

Any other error code set in the `error` parameter in the response will be treated by the Availability Management Framework as if the caller had set the `error` parameter to `SA_AIS_ERR_FAILED_OPERATION`.

If the invoked process does not respond to this callback or responds to it by calling `saAmfResponse_4()` with the `error` parameter set to `SA_AIS_ERR_FAILED_OPERATION` within the time interval configured by the value of the `saAmfCompCleanupTimeout` configuration attribute of the `SaAmfComp` configuration object class (see Section 8.13.2) for the component referred to by the `containedCompName` parameter, the Availability Management Framework must engage the configured recovery policy (see Section 3.11.1.3) for the container component.

See Also

`saAmfResponse_4()`, `saAmfComponentRegister()`, `saAmfDispatch()`, `SaAmfContainedComponentInstantiateCallbackT`

7.11 Protection Group Management

7.11.1 saAmfProtectionGroupTrack_4()

Prototype

```
SaAisErrorT saAmfProtectionGroupTrack_4(
      SaAmfHandleT amfHandle,
      const SaNameT *csiName,
      SaUint8T trackFlags,
      SaAmfProtectionGroupNotificationBufferT_4
            *notificationBuffer
);
```

Parameters

`amfHandle` - [in] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`csiName` - [in] A pointer to the name of the component service instance for which tracking is to start. This name is also the name of the protection group. The `SaNameT` type is defined in [2].

`trackFlags` - [in] The kind of tracking that is requested, which is the bitwise OR of one or more of the following flags (as defined in [2]), which have the following interpretation here:

- SA_TRACK_CURRENT - If `notificationBuffer` is NULL, information about all components in the protection group is returned by a single subsequent invocation of the `saAmfProtectionGroupTrackCallback()` notification callback; otherwise, this information is returned in the structure to which `notificationBuffer` points when the `saAmfProtectionGroupTrack_4()` call completes successfully.
- SA_TRACK_CHANGES - The notification callback is invoked each time at least one change occurs in the protection group membership, or one attribute (HA state, HA readiness state, or rank) of at least one component in the protection group changes. Information about all of the components is passed to the callback.

- SA_TRACK_CHANGES_ONLY - The notification callback is invoked each time at least one change occurs in the protection group membership, or one attribute (HA state, HA readiness state, or rank) of at least one component in the protection group changes. Only information about components in the protection group that have changed is passed to this callback function.

It is not permitted to set both SA_TRACK_CHANGES and SA_TRACK_CHANGES_ONLY in an invocation of this function. The SaUint8T type is defined in [2].

notificationBuffer - [in/out] - A pointer to a structure of type SaAmfProtectionGroupNotificationBufferT_4 (defined in Section 7.4.6.4 on page 256). This parameter is ignored if SA_TRACK_CURRENT is not set in trackFlags; otherwise and notificationBuffer is not NULL, the structure will contain information about all components in the protection group when saAmfProtectionGroupTrack_4() returns. The meaning of the fields of the SaAmfProtectionGroupNotificationBufferT_4 structure is:

- numberOfItems - [in/out] If notification is NULL, numberOfItems is ignored as input parameter; otherwise, it specifies that the array pointed to by notification provides memory for information about numberOfItems components in the protection group.
  When saAmfProtectionGroupTrack_4() returns with SA_AIS_OK or with SA_AIS_ERR_NO_SPACE, numberOfItems contains the number of components in the protection group.
- notification - [in/out] If notification is NULL, memory for the protection group information is allocated by the Availability Management Framework. The caller is responsible for freeing the allocated memory by calling the saAmfProtectionGroupNotificationFree_4() function.

Description

The Availability Management Framework is requested to start tracking changes in the protection group associated with the component service instance identified by the name to which csiName points or changes of attributes of any component in the protection group. These changes are notified by the invocation of the saAmfProtectionGroupTrackCallback() callback function, which must have been supplied when the process invoked the saAmfInitialize_4() call.

An application may call saAmfProtectionGroupTrack_4() repeatedly for the same values of amfHandle and of the component service instance designated by the name referred to by csiName, regardless of whether the call initiates a one-time status request or a series of callback notifications.

If saAmfProtectionGroupTrack_4() is called with trackFlags containing SA_TRACK_CHANGES_ONLY while changes in the protection group are currently being tracked with SA_TRACK_CHANGES for the same combination of amfHandle and the component service instance designated by the name referred to by csiName, the Availability Management Framework will invoke further notification callbacks according to the new trackFlags. The same is true vice versa.
Once saAmfProtectionGroupTrack_4() has been called with trackFlags containing either SA_TRACK_CHANGES or SA_TRACK_CHANGES_ONLY, notification callbacks can only be stopped by an invocation of saAmfProtectionGroupTrackStop().

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle amfHandle is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INIT - The previous invocation of saAmfInitialize_4() to initialize the Availability Management Framework was incomplete, as the saAmfProtectionGroupTrackCallback() callback function is missing. This value is not returned if only the SA_TRACK_CURRENT flag is set in trackFlags and the notificationBuffer parameter is not NULL.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NO_SPACE - The SA_TRACK_CURRENT flag is set, and the notification pointer in the structure pointed to by notificationBuffer is not NULL, but the value of numberOfItems in this structure is smaller than the number of entries to be provided in the array referred to by the notification pointer.

SA_AIS_ERR_NOT_EXIST - The component service instance designated by the name referred to by csiName cannot be found.

SA_AIS_ERR_BAD_FLAGS - The `trackFlags` parameter is invalid.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaAmfProtectionGroupTrackCallbackT_4`, `saAmfProtectionGroupTrackStop()`, `saAmfProtectionGroupNotificationFree_4()`, `saAmfInitialize_4()`

7.11.2 SaAmfProtectionGroupTrackCallbackT_4

Prototype

```
typedef void (*SaAmfProtectionGroupTrackCallbackT_4)(
     const SaNameT *csiName,
     SaAmfProtectionGroupNotificationBufferT_4
          *notificationBuffer,
     SaUint32T numberOfMembers,
     SaAisErrorT error
);
```

Parameters

`csiName` - [in] A pointer to the name of the component service instance. The `SaNameT` type is defined in [2].

`notificationBuffer` - [in] A pointer to a structure to contain the requested information about components in the protection group. The `SaAmfProtectionGroupNotificationBufferT_4` is defined in Section 7.4.6.4 on page 256.

`numberOfMembers` - [`in`] The number of the components that belong to the protection group associated with the component service instance designated by the name to which `csiName` refers. The `SaUint32T` type is defined in [2].

`error` - [`in`] This parameter indicates whether the Availability Management Framework was able to perform the operation. Possible values for the `error` parameter (whose type is defined in [2]) are:

- `SA_AIS_OK` - The function completed successfully.
- `SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.
- `SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.
- `SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` that was passed to the corresponding `saAmfProtectionGroupTrack_4()` call has become invalid, since it is corrupted, uninitialized, or has already been finalized.
- `SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service. The process that invoked `saAmfProtectionGroupTrack_4()` might have missed one or more notifications.
- `SA_AIS_ERR_NO_RESOURCES` - Either the Availability Management Framework library or the provider of the service is out of required resources (other than memory), and cannot provide the service. The process that invoked `saAmfProtectionGroupTrack_4()` might have missed one or more notifications.
- `SA_AIS_ERR_NOT_EXIST` - The component service instance designated by the name referred to by `csiName` has been administratively deleted.
- `SA_AIS_ERR_UNAVAILABLE` - The operation requested in the corresponding `saAmfProtectionGroupTrack_4()` call is unavailable on this cluster node due to one of the two reasons:
  - the cluster node has left the cluster membership;
  - the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

If the error returned is `SA_AIS_ERR_NO_MEMORY` or `SA_AIS_ERR_NO_RESOURCES`, the process that invoked `saAmfProtectionGroupTrack_4()` should invoke `saAmfProtectionGroupTrackStop()` and then invoke `saAmfProtectionGroupTrack_4()` again to resynchronize with the current state.

Description

This callback is invoked in the context of a thread calling `saAmfDispatch()` with the handle `amfHandle` that was specified when the process called `saAmfProtectionGroupTrack_4()` to request tracking of changes in the protection group associated with the component service instance identified by the name to which `csiName` refers or in an attribute of any component in this protection group.

If successful, the `saAmfProtectionGroupTrackCallback()` function returns the requested information in the structure pointed to by the `notificationBuffer` parameter. The kind of information returned depends on the setting of the `trackFlags` parameter of the `saAmfProtectionGroupTrack_4()` function.

The value of the `numberOfItems` attribute in the structure to which the `notificationBuffer` parameter points might be greater than the value of the `numberOfMembers` parameter, because some components may no longer be members of the protection group: if the `SA_TRACK_CHANGES` flag or the `SA_TRACK_CHANGES_ONLY` flag is set, the structure to which `notificationBuffer` points might contain information about the current members of the protection group and also about components that have recently left the protection group.

If an error occurs, it is returned in the error parameter.

Return Values

None

See Also

`saAmfProtectionGroupTrack_4()`, `saAmfProtectionGroupTrackStop()`, `saAmfDispatch()`

7.11.3 saAmfProtectionGroupTrackStop()

Prototype

```
SaAisErrorT saAmfProtectionGroupTrackStop(
      SaAmfHandleT amfHandle,
      const SaNameT *csiName
);
```

Parameters

`amfHandle` - [in] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`csiName` - [in] A pointer to the name of the component service instance. The `SaNameT` type is defined in [2].

Description

The invoking process requests the Availability Management Framework to stop tracking protection group changes for the component service instance identified by the name to which `csiName` points.

The Availability Management Framework releases any resources that it allocated for the tracking to be stopped.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - This value is returned if one or both cases below occurred.

- The component service instance designated by the name referred to by csiName cannot be found.
- No track of protection group changes for the component service instance designated by the name referred to by csiName was previously started by invoking saAmfProtectionGroupTrack_4() with track flags SA_TRACK_CHANGES or SA_TRACK_CHANGES_ONLY that would still be in effect.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle amfHandle was acquired before the cluster node left the cluster membership.

See Also

SaAmfProtectionGroupTrackCallbackT_4, saAmfProtectionGroupTrack_4(), saAmfInitialize_4()

7.11.4 saAmfProtectionGroupNotificationFree_4()

Prototype

```
SaAisErrorT saAmfProtectionGroupNotificationFree_4(
      SaAmfHandleT amfHandle,
      SaAmfProtectionGroupNotificationT_4 *notification
);
```

Parameters

amfHandle - [in] The handle which was obtained by a previous invocation of the saAmfInitialize_4() function and which identifies this particular initialization of the Availability Management Framework. The SaAmfHandleT type is defined in Section 7.4.1 on page 245.

`notification` - [in] A pointer to the memory that was allocated by the Availability Management Framework library in the `saAmfProtectionGroupTrack_4()` function and is to be released. The `SaAmfProtectionGroupNotificationT_4` type is defined in Section 7.4.6.3 on page 256.

Description

This function frees the memory to which `notification` points and which was allocated by the Availability Management Framework library in a previous call to the `saAmfProtectionGroupTrack_4()` function.

For details, refer to the description of the `notification` pointer in the structure referred to by the `notificationBuffer` parameter in the corresponding invocation of the `saAmfProtectionGroupTrack_4()` function.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfProtectionGroupTrack_4()`

7.12 Error Reporting

7.12.1 saAmfComponentErrorReport_4()

Prototype

```
SaAisErrorT saAmfComponentErrorReport_4(
      SaAmfHandleT amfHandle,
      const SaNameT *erroneousComponent,
      SaTimeT errorDetectionTime,
      SaAmfRecommendedRecoveryT recommendedRecovery,
      SaNtfCorrelationIdsT *correlationIds
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`erroneousComponent` - [`in`] A pointer to the name of the erroneous component. The `SaNameT` type is defined in [2].

`errorDetectionTime` - [`in`] The absolute time when the reporting component detected the error. If this value is 0, it is assumed that the time at which the library received the error is the error detection time. The `SaTimeT` type is defined in [2].

`recommendedRecovery` - [`in`] Recommended recovery action. The `SaAmfRecommendedRecoveryT` type is defined in Section 7.4.7 on page 257.

`correlationIds` - [`in/out`] Pointer to correlation identifiers associated with the error report.The `rootCorrelationId` and `parentCorrelationId` fields are `in` parameters and hold the root and parent correlation identifiers, respectively. These correlation identifiers are included by the Availability Management Framework in its own notifications triggered by this error report. The `rootCorrelationId` and `parentCorrelationId` may hold the same value. If these notification identifiers are not available to the invoker of this function, the correlation identifiers must be set to `SA_NTF_IDENTIFIER_UNUSED`. The Availability Management Framework returns in the `notificationId` field the identifier of the error report notification it sends itself when it receives this error report. The `SaNtfCorrelationIdsT` type is

defined
in [3].

Description

The `saAmfComponentErrorReport_4()` function reports an error and provides a recovery recommendation to the Availability Management Framework. The Availability Management Framework validates the recommended recovery action and reacts to it as described in Section 3.11.2.1 on page 201.

The `correlationIds` parameter is used by a caller process to inform the Availability Management Framework about any notifications generated previously and which can be correlated with the error condition being reported.

If the Availability Management Framework has already generated any error report notifications for the same error condition (for instance, an error has already been reported for the same component, and it has not yet been cleared), the Availability Management Framework shall include the identifiers of those notifications in the error report notification generated as a result of the current invocation. Any such notification identifiers are included as correlated sibling notifications, that is, they are placed at the third and higher positions of the correlated notifications attribute of the error report notification (see Section 11.2.3.1).

Return Values

`SA_AIS_OK` - The function returned successfully, and the Availability Management Framework has been notified of the error report.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly. In particular, this value is returned if the `SA_AMF_CONTAINER_RESTART`: recommended recovery is set in `recommendedRecovery`, and `erroneousComponent` does not point to the name of a contained component.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - The component specified by the name to which `erroneousComponent` refers is not contained in the Availability Management Framework's configuration.

SA_AIS_ERR_ACCESS - The Availability Management rejects the requested recommended recovery.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saAmfComponentErrorClear_4()`, `saAmfInitialize_4()`

7.12.2 saAmfComponentErrorClear_4()

Prototype

```
SaAisErrorT saAmfComponentErrorClear_4(
      SaAmfHandleT amfHandle,
      const SaNameT *compName,
      SaNtfCorrelationIdsT *correlationIds
);
```

Parameters

`amfHandle` - [in] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`compName` - [in] A pointer to the name of the component to be cleared of all errors. The `SaNameT` type is defined in [2].

`correlationIds` - [in/out] Pointer to correlation identifiers associated with the error clear. The `rootCorrelationId` and `parentCorrelationId` fields are `in` parameters and hold the root and parent correlation identifiers, respectively. These correlation identifiers are included by the Availability Management Framework in its own notifications triggered by this error clear. The `rootCorrelationId` and `parentCorrelationId` may hold the same value. If these notification identifiers are not available to the invoker of this function, the correlation identifiers must be set to `SA_NTF_IDENTIFIER_UNUSED`. The Availability Management Framework returns in the `notificationId` field the identifier of the error clear notification it sends itself when it receives this error clear. The `SaNtfCorrelationIdsT` type is defined in [3].

Description

This function cancels the previous errors reported for the component identified by the name to which `compName` refers. This function indicates the availability of a component for providing service after an externally executed repair action and results in enabling the operational state of the component.

If the repaired component was the only component left with a disabled operational state in its service unit, the Availability Management Framework also sets the operational state of the service unit containing the repaired component to enabled.

A component enters the disabled operational state due to the reasons stated in Section 3.2.2.2 on page 75.

The Availability Management Framework typically engages in repairing the component. However, if this repair fails, or such repair actions are not permitted by the configuration, this function can be used by an external entity to indicate that the component has been repaired and its operational state is now enabled.

The Availability Management Framework expects that a repair done by an external entity brings the repaired component to a state equivalent to the uninstantiated presence state, before this function is invoked. Thus, the Availability Management Framework sets in the information model the presence state attribute of the component and of the enclosing service unit to uninstantiated and clears any instantiation-failed or termination-failed error condition.

The `correlationIds` parameter is used by a caller process to inform the Availability Management Framework about any notifications generated previously and which can be correlated with the error condition being cleared.

The Availability Management Framework shall generate only one error clear notification for each error condition. That is, if the error condition for the components has already been cleared, it must not generate a second error clear notification. In the generated error clear notification, the Availability Management Framework shall include the identifiers of all error report notifications it has generated for the error con-

dition being cleared. Any such notification identifiers are included as correlated sibling notifications, that is, they are placed at the third and higher positions of the correlated notifications attribute of the error clear notification (see Section 11.2.3.2).

This function can be invoked on a component hosted by an AMF node even if this AMF node is not mapped to a CLM node, or if the underlying CLM node is not a member node. It can also be issued on a component even if it is configured but uninstantiated.

If this function is invoked on a component whose operational state is already enabled, the component remains in that state, and a benign error value of `SA_AIS_ERR_NO_OP` is returned to the caller.

Return Values

`SA_AIS_OK` - The function returned successfully, and the Availability Management Framework has been reliably notified about clearing the error.
Upon return, it is guaranteed that the Availability Management Framework will not lose the error clear instruction, as long as the cluster is not reset.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - The system is out of required resources (other than memory).

`SA_AIS_ERR_NOT_EXIST` - The component specified by the name to which `compName` points is not contained in the Availability Management Framework's configuration.

`SA_AIS_ERR_NO_OP` - The invocation of this function has no effect on the current operational state of the component specified by the name to which `compName` points, as the operational state of the component is already enabled.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle amfHandle was acquired before the cluster node left the cluster membership.

See Also

saAmfComponentErrorReport_4(), saAmfInitialize_4()

7.12.3 saAmfCorrelationIdsGet()

Prototype

```
SaAisErrorT saAmfCorrelationIdsGet(
      SaAmfHandleT amfHandle,
      SaInvocationT invocation,
      SaNtfCorrelationIdsT *correlationIds
);
```

Parameters

amfHandle - [in] The handle which was obtained by a previous invocation of the saAmfInitialize_4() function and which identifies this particular initialization of the Availability Management Framework. The SaAmfHandleT type is defined in Section 7.4.1 on page 245.

invocation - [in] This parameter identifies a particular invocation of a callback function by the Availability Management Framework. The SaInvocationT type is defined in [2].

correlationIds - [out] A pointer to the correlation identifiers to which the invocation of the callback function identified by invocation is related. The SaNtfCorrelationIdsT type is defined in [3].

Description

This function is typically used by a process that needs to generate a notification as a consequence of a particular callback invocation that the process has received from the Availability Management Framework, and the process wants to set the proper correlation identifiers in this notification to allow the reconstruction of the notification correlation tree.

The `invocation` parameter refers to the particular callback invocation made by the Availability Management Framework, and it can be used as long as it has not been invalidated yet

- by the process
  - when it responds to the callback by invoking the `saAmfResponse_4()` function or
  - when it calls the `saAmfCSIQuiescingComplete()` function to inform the Availability Management Framework that its component has completed the quiescing of services, or
- by the Availability Management Framework when it interrupts the quiescing of services of the component to which the process belongs and sets another HA state for these component service instances. The quiescing is requested by the Availability Management Framework when it invokes the `saAmfCSIQuiescingComplete()` callback function.

The Availability Management Framework returns in the `rootCorrelationId` and `parentCorrelationId` fields of the structure referred to by the `correlationIds` pointer the notification identifiers of the root and parent notifications that shall be used by the process in the generated notifications related to the invocation of the callback function identified by the `invocation` parameter. The notificationId field is not used.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - The callback invocation identified by the `invocation` parameter does not identify a currently in-progress AMF callback.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaAmfComponentTerminateCallbackT`, `SaAmfCSISetCallbackT`, `SaAmfCSIRemoveCallbackT`, `SaAmfProxiedComponentInstantiateCallbackT`, `SaAmfProxiedComponentCleanupCallbackT`, `saAmfComponentRegister()`, `saAmfInitialize_4()`, `SaAmfContainedComponentInstantiateCallbackT`, `SaAmfContainedComponentCleanupCallbackT`

7.13 Component Response to Framework Requests

7.13.1 saAmfResponse_4()

Prototype

```
SaAisErrorT saAmfResponse_4(
      SaAmfHandleT amfHandle,
      SaInvocationT invocation,
      SaNtfCorrelationIdsT *correlationIds,
      SaAisErrorT error
);
```

Parameters

`amfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saAmfInitialize_4()` function and which identifies this particular initialization of the Availability Management Framework. The `SaAmfHandleT` type is defined in Section 7.4.1 on page 245.

`invocation` - [`in`] This parameter associates an invocation of this response function with a particular invocation of a callback function by the Availability Management Framework. The `SaInvocationT` type is defined in [2].

`correlationIds` - [`in/out`] Pointer to correlation identifiers associated with the response if the response is reporting an error; otherwise, this pointer is set to NULL. The `rootCorrelationId` and `parentCorrelationId` fields are `in` parameters and hold the root and parent correlation identifiers, respectively. These correlation identifiers are included by the Availability Management Framework in its own notifications triggered by this response. The `rootCorrelationId` and `parentCorrelationId` fields may hold the same value. The Availability Management Framework returns in the `notificationId` field the identifier of the error report notification it sends itself when it receives this response. The `SaNtfCorrelationIdsT` type is defined in [3].

`error` - [`in`] The response of the process to the associated callback. It returns `SA_AIS_OK` if the associated callback was successfully executed by the process; otherwise, it returns an appropriate error, as described in the corresponding callback. The `SaAisErrorT` type is defined in [2].

Description

The component responds to the Availability Management Framework with the result of the execution of an operation that was requested by the Availability Management Framework when it invoked a callback specifying `invocation` to identify the requested operation. In the `saAmfResponse_4()` call, the component gives that value of `invocation` back to the Availability Management Framework, so that the Availability Management Framework can associate this response with the callback request.

The request can be one of the following types.

- Request for executing a given healthcheck. See `SaAmfHealthcheckCallbackT`.
- Request for terminating a component. See `SaAmfComponentTerminateCallbackT`.
- Request for adding/assigning a given HA state to a component on behalf of a component service instance. See `SaAmfCSISetCallbackT`.
- Request for removing a component service instance from a component. See `SaAmfCSIRemoveCallbackT`.
- Request for instantiating a proxied component. See `SaAmfProxiedComponentInstantiateCallbackT`.
- Request for cleaning up a proxied component. See `SaAmfProxiedComponentCleanupCallbackT`.
- Request for instantiating a contained component. See `SaAmfContainedComponentInstantiateCallbackT`.
- Request for cleaning up a contained component. See `SaAmfContainedComponentCleanupCallbackT`.

The component replies to the Availability Management Framework when either (i) it cannot carry out the request, or (ii) it has failed to successfully complete the execution of the request, or (iii) it has successfully completed the request. In cases (i) and (ii), the responding process uses the `correlationIds` parameter to indicate known correlated notifications. In case (iii), this parameter must be set to NULL.

With the exception of the response to an `saAmfHealthcheckCallback()` call, this function may be called only by a registered process for a component, that is, the `amfHandle` must be the same that was used when the registered process registered the component by invoking `saAmfComponentRegister()`. The response to an `saAmfHealthcheckCallback()` call may only be issued by the process that started this healthcheck.

Return Values

SA_AIS_OK - The function returned successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `amfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - The `invocation` parameter does not identify a callback that has an outstanding response.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Availability Management Framework library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `amfHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaAmfHealthcheckCallbackT`, `SaAmfComponentTerminateCallbackT`, `SaAmfCSISetCallbackT`, `SaAmfCSIRemoveCallbackT`, `SaAmfProxiedComponentInstantiateCallbackT`, `SaAmfProxiedComponentCleanupCallbackT`, `saAmfComponentRegister()`, `saAmfInitialize_4()`, `SaAmfContainedComponentInstantiateCallbackT`, `SaAmfContainedComponentCleanupCallbackT`

8 AMF UML Information Model

The Availability Management Framework information model is described in UML and has been organized in UML class diagrams.

The Availability Management Framework UML model is implemented by the SA Forum IMM Service ([6]). For further details on this implementation, refer to the SA Forum Overview document ([1]).

The classes in the Availability Management Framework UML class diagrams show the contained attributes and their type, multiplicity, default values, and constraints. The description of each attribute is provided in the SA Forum XMI document (see [7]). The class diagrams additionally show the administrative operations (if any) applicable on these classes.

To simplify references, this description uses for the UML diagrams the same names used in [7].

The UML diagrams defined for the Availability Management Framework are:

- “3- Cluster View”
- “3.1- AMF Instances and Types View”
- “3.2- AMF Instances View”
- “3.3- AMF Cluster, Node, and Node Group Classes”
- “3.4- AMF Application Classes”
- “3.5- AMF SG Classes”
- “3.6- AMF SU Classes”
- “3.7- AMF SI Classes”
- “3.8- AMF CSI Classes”
- “3.9a- AMF Component Classes”
- “3.9b- AMF Component Type Classes”
- “3.9c- AMF Global Component Attributes and Healthcheck Classes”

These diagrams will be described starting with Section 8.4.

8.1 Use of Entity Types in the AMF UML Information Model

As has been described in Chapter 3, types are defined for several entities of the Availability Management Framework. This is done for the purpose of facilitating the configuration and for software management purposes such as upgrading a configuration (see [8]).

Types containing a version represent a generalization of entities.

Entity types that differ only by their version are grouped together into a base entity type. Thus, the base entity type typically reflects some common functionality and features that are common to all entities of the type regardless of their version. As a result, a base entity type is not directly associated with code or other executables, and no instances of it exist at runtime.

The attributes of an entity object class are in a particular relation with the attributes of the corresponding entity type object class. An attribute of the entity type object `saAmf<entity type>Def<attribute name>` defines the default value for the `saAmf<entity><attribute>` attribute of the appropriate entity objects. The `saAmf<entity><attribute>` attribute of an entity object class may override or complement the default value provided by the `saAmf<entity type>Def<attribute name>` attribute of the entity type object as follows:

- A value specified for an attribute `saAmf<entity><attribute>` overrides the value of the associated `saAmf<entity type>Def<attribute name>` attribute if the `saAmf<entity type>Def<attribute name>` attribute is specified as a default value for `saAmf<entity><attribute>` attribute in the entity class. If the `<attribute name>` contains the "Min" or "Max" tag, the `saAmf<entity type>Def<attribute name>` value sets the lower/upper boundary for the overriding value.
- In any other case, the `saAmf<entity><attribute>` value complements the value specified by `saAmf<entity type>Def<attribute name>`.

8.2 Notes on the Conventions Used in UML Diagrams

A general explanation of the conventions used in the UML diagrams, such as the use of constraints, default values, and the like is presented in [1].

8.3 DN Formats for Availability Management Framework UML Classes

Table 21 provides the format of the various DNs used to name Availability Management Framework objects of the SA Forum Information Model. One format is defined for each object class. The '*' notation at the end of a DN format indicates that none or more RDNs may be appended to the proposed format.

Table 21 DN Formats

Object Class	DN Format for Objects of that Class
SaAmfAppBaseType	"safAppType=..."
SaAmfApplication	"safApp=..."
SaAmfAppType	"safVersion=...,safAppType=..."
SaAmfCluster	"safAmfCluster=..."
SaAmfComp	"safComp=...,safSu=...,safSg=...,safApp=..."
SaAmfCompBaseType	"safCompType=..."
SaAmfCompCsType	"safSupportedCsType=...,safComp=...,safSu=...,safSg=...,safApp=..."
SaAmfCompGlobalAttributes	"safRdn=compGlobalAttributes, safApp=safAmfService"
SaAmfCompType	"safVersion=...,safCompType=..."
SaAmfCSBaseType	"safCSType=..."
SaAmfCSI	"safCsi=...,safSi=...,safApp=..."
SaAmfCSIAssignment	"safCSIComp=...,safCsi=...,safSi=...,safApp=..."
SaAmfCSIAttribute	"safCsiAttr=...,safCsi=...,safSi=...,safApp=..."
SaAmfCSType	"safVersion=...,safCSType=..."
SaAmfCtCsType	"safSupportedCsType=...,safVersion=...,safCompType=..."
SaAmfHealthcheck (instance)	"safHealthcheckKey=...,safComp=...,safSu=...,safSg=...,safApp=
SaAmfHealthcheck (type)	"safHealthcheckKey=...,safVersion=...,safCompType=..."
SaAmfNode	"safAmfNode=...,safAmfCluster=..."
SaAmfNodeGroup	"safAmfNodeGroup=...,safAmfCluster=..."
SaAmfNodeSwBundle	"safInstalledSwBundle=..., safAmfNode=...,safAmfCluster=..."
SaAmfSG	"safSg=...,safApp=..."

Table 21 DN Formats (Continued)

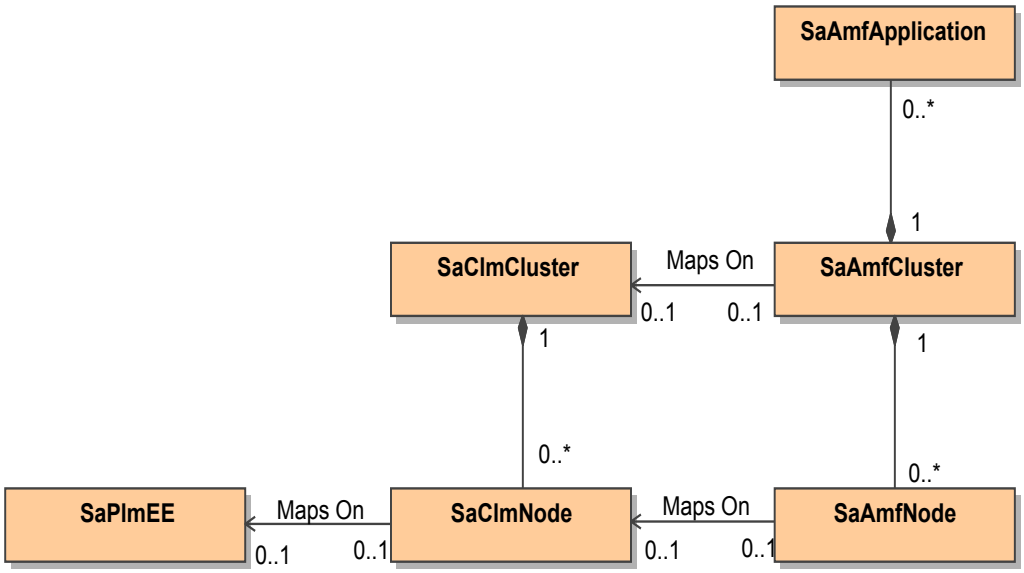
Object Class	DN Format for Objects of that Class
SaAmfSGBaseType	"safSgType=..."
SaAmfSGType	"safVersion=...,safSgType=..."
SaAmfSI	"safSi=...,safApp=..."
SaAmfSIAssignment	"safSISU=...,safSi=...,safApp=..."
SaAmfSIDependency	"safDepend=...,safSi=...,safApp=..."
SaAmfSIRankedSU	"safRankedSu=...,safSi=...,safApp=..."
SaAmfSU	"safSu=...,safSg=...,safApp=..."
SaAmfSUBaseType	"safSuType=..."
SaAmfSutCompType	"safMemberCompType=...,safVersion=...,safSuType=..."
SaAmfSUType	"safVersion=...,safSuType=..."
SaAmfSvcBaseType	"safSvcType=..."
SaAmfSvcType	"safVersion=...,safSvcType=..."
SaAmfSvcTypeCSTypes	"safMemberCSType=...,safVersion=...,safSvcType=..."

8.4 AMF Cluster

The following overview picture shows the relationships among the classes `SaPlmEE`, `SaClmCluster`, `SaClmNode`, and the Availability Management Framework classes `SaAmfCluster`, `SaAmfNode`, and `SaAmfApplication`.

Attributes and operations of the Availability Management Framework classes `SaAmfCluster` and `SaAmfNode` are shown in Section 8.7. Attributes and operations of the Availability Management Framework `SaAmfApplication` class are shown in Section 8.8. The classes `SaClmCluster` and `SaClmNode` are described in [4], and the `SaPlmEE` class is described in [5].

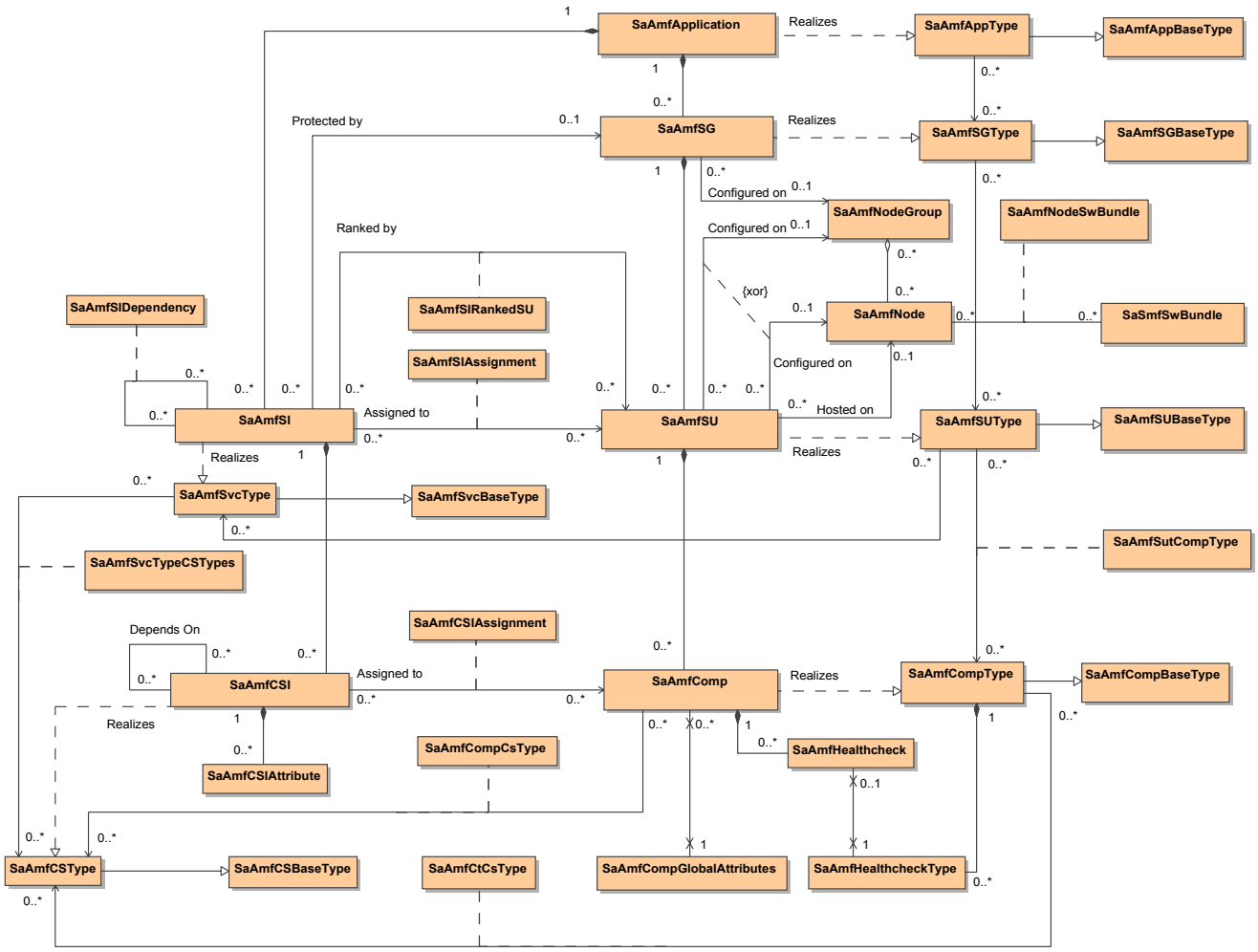
FIGURE 27 3- Cluster View



8.5 Availability Management Framework Instances and Types View

This overview diagram shows all classes of the Availability Management Framework, except the `SaAmfCluster` class, which is shown in Section 8.7.

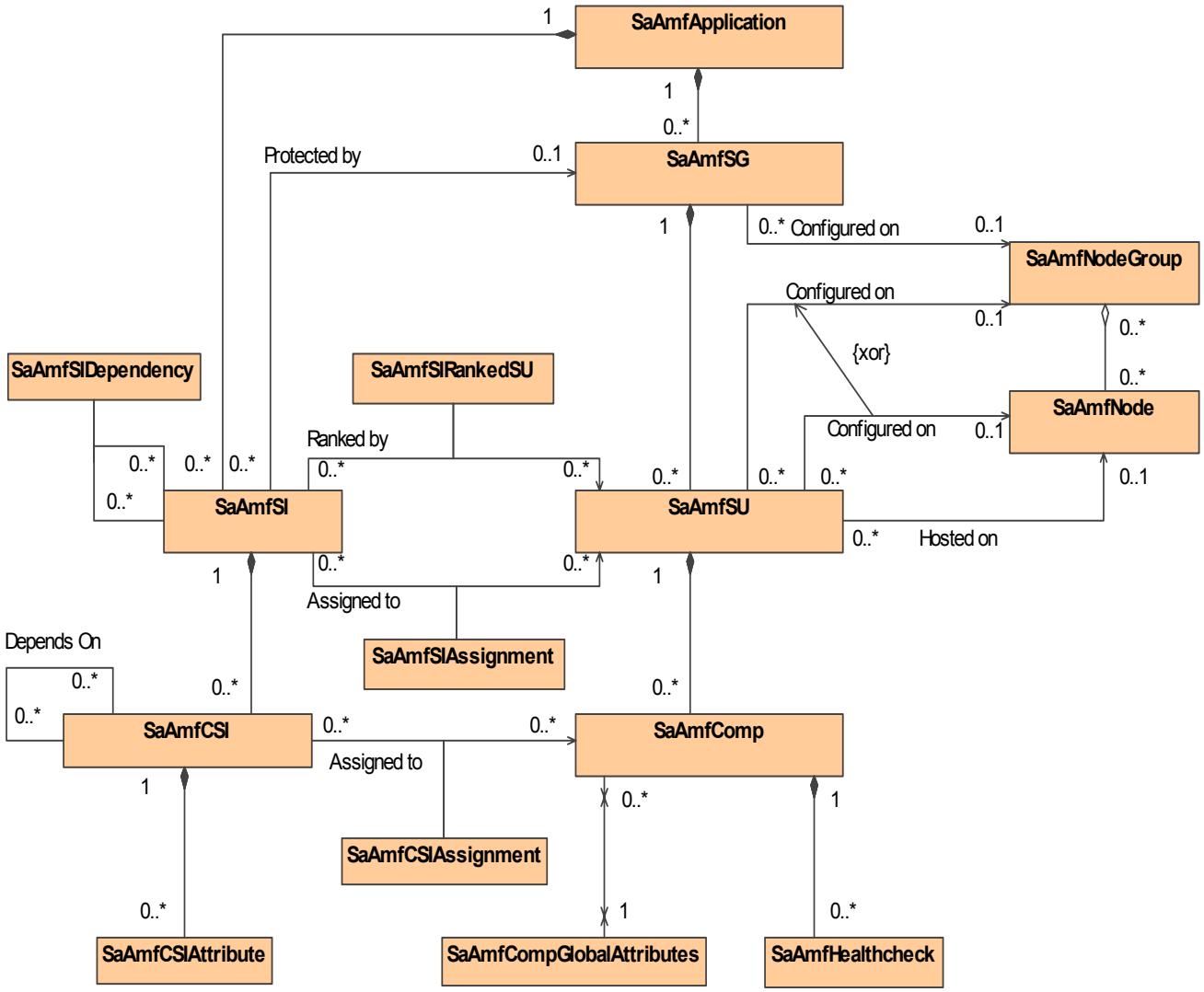
FIGURE 28 3.1- AMF Instances and Types View



8.6 Availability Management Framework Instances View

This overview diagram simplifies the overview of Section 8.5 by omitting all type classes.

FIGURE 29 3.2- AMF Instances View

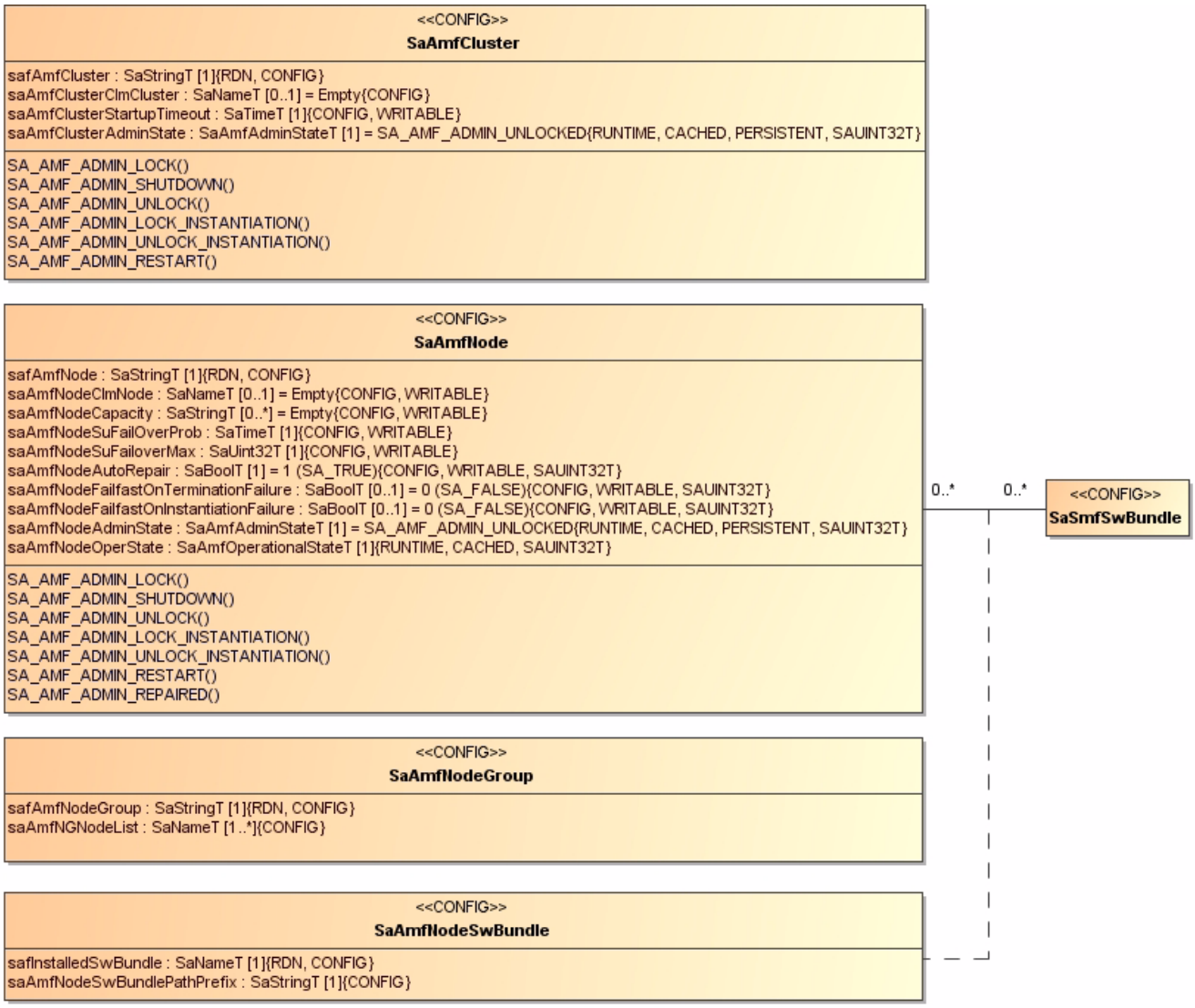


8.7 AMF Cluster, Node, and Node-Related Classes

The classes of this diagram are:

- `SaAmfCluster` — This configuration object class defines the configuration and runtime attributes of an AMF cluster and the operations that can be applied on the AMF cluster. An object of this class must be configured for each AMF cluster. For details, refer to Section 3.1.1 on page 38, Section 3.2.8 on page 93, and Chapter 9.
- `SaAmfNode` — This configuration object class defines the configuration and runtime attributes of an AMF node and the operations that can be applied on the AMF node. An object of this class must be configured for each AMF node. For details, refer to Section 3.1.1 on page 38, Section 3.2.6 on page 90, Section 3.6.1.3 on page 114, and Chapter 9.
- `SaAmfNodeGroup` — This configuration object class defines the configuration attributes of a node group, which is used in the configuration of service groups and service units to specify AMF nodes that can host these entities. An object of this class can be configured for a local service unit or a service group that has local service units. For further details, refer to Section 3.1.9 on page 57, Section 8.9 on page 348, and Section 8.10 on page 350. No administrative operations are defined for a node group.
- `SaAmfNodeSwBundle` — This is a configuration association class between the `SaAmfNode` and `SaSmfSwBundle` object classes. The `SaAmfNodeSwBundle` class defines the root installation directory of a particular software bundle on the AMF node in question. It is used to determine the absolute CLC-CLI command path for components of component types delivered by the software bundle when such a component is mapped onto the AMF node.

FIGURE 30 3.3- AMF Cluster, Node, and Node-Related Classes

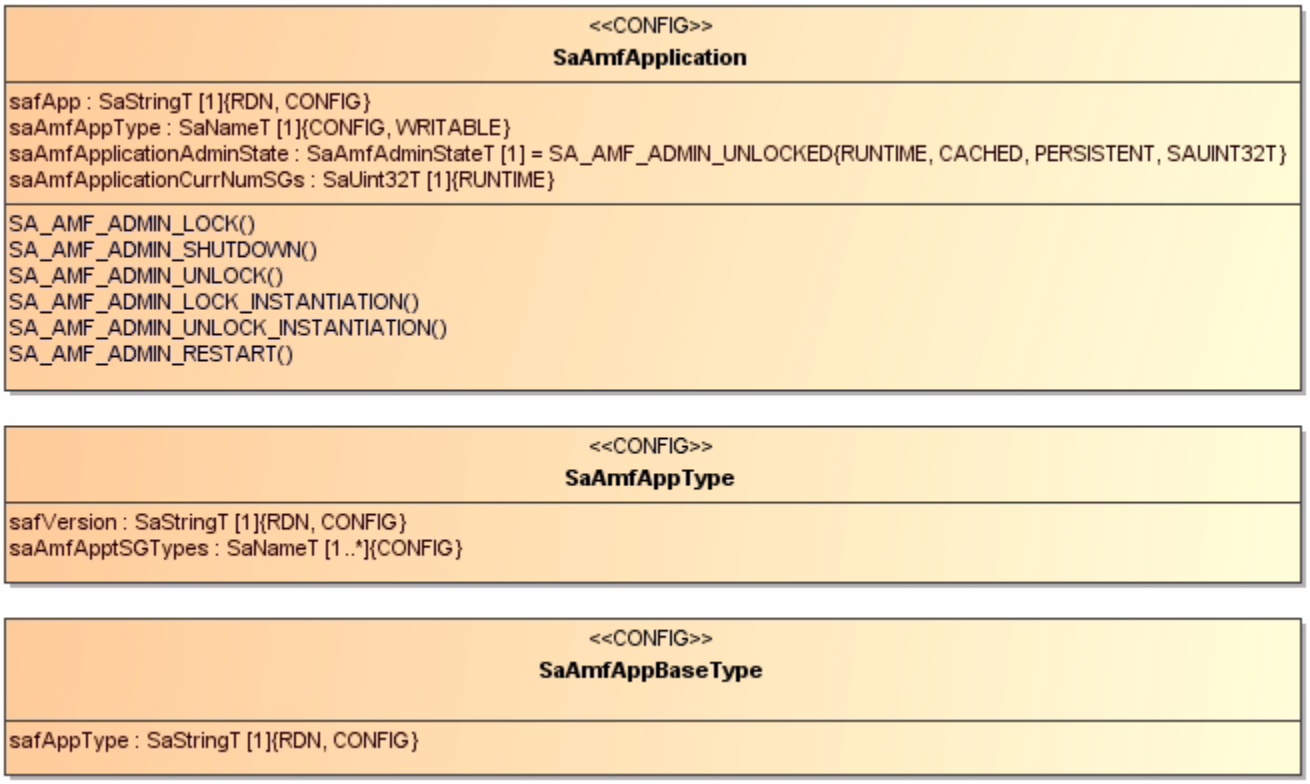


8.8 Application Classes Diagram

The classes of this diagram are:

- `SaAmfApplication` — This configuration object class defines configuration and runtime attributes of an application and the operations that can be applied on the application. For each application, an object of this class must be configured, and its `saAmfApplicationType` attribute must contain the DN of a valid object of the `SaAmfApplicationType` object class. Additional configuration attributes of an application are defined in the `SaAmfAppType` class.
- `SaAmfAppType` — This configuration object class defines configuration attributes of an application type. An application type defines a list of service group types, which implies that an application of the given type must be composed of service groups of types from that list. All applications of the same type share the attribute values defined in the application type configuration. Some of the attribute values of the application type may be overridden in the configuration of any application by setting the corresponding attribute of the configuration object of the application to the required value.
- `SaAmfAppBaseType` — This configuration object class defines the configuration attributes common to different application types. In particular, a base application type defines the common name of versioned application types. An application type x belongs to a base application type y based on the DN of x, which is the concatenation of the RDN of x (representing its version) with the DN of y.

FIGURE 31 3.4- AMF Application Classes

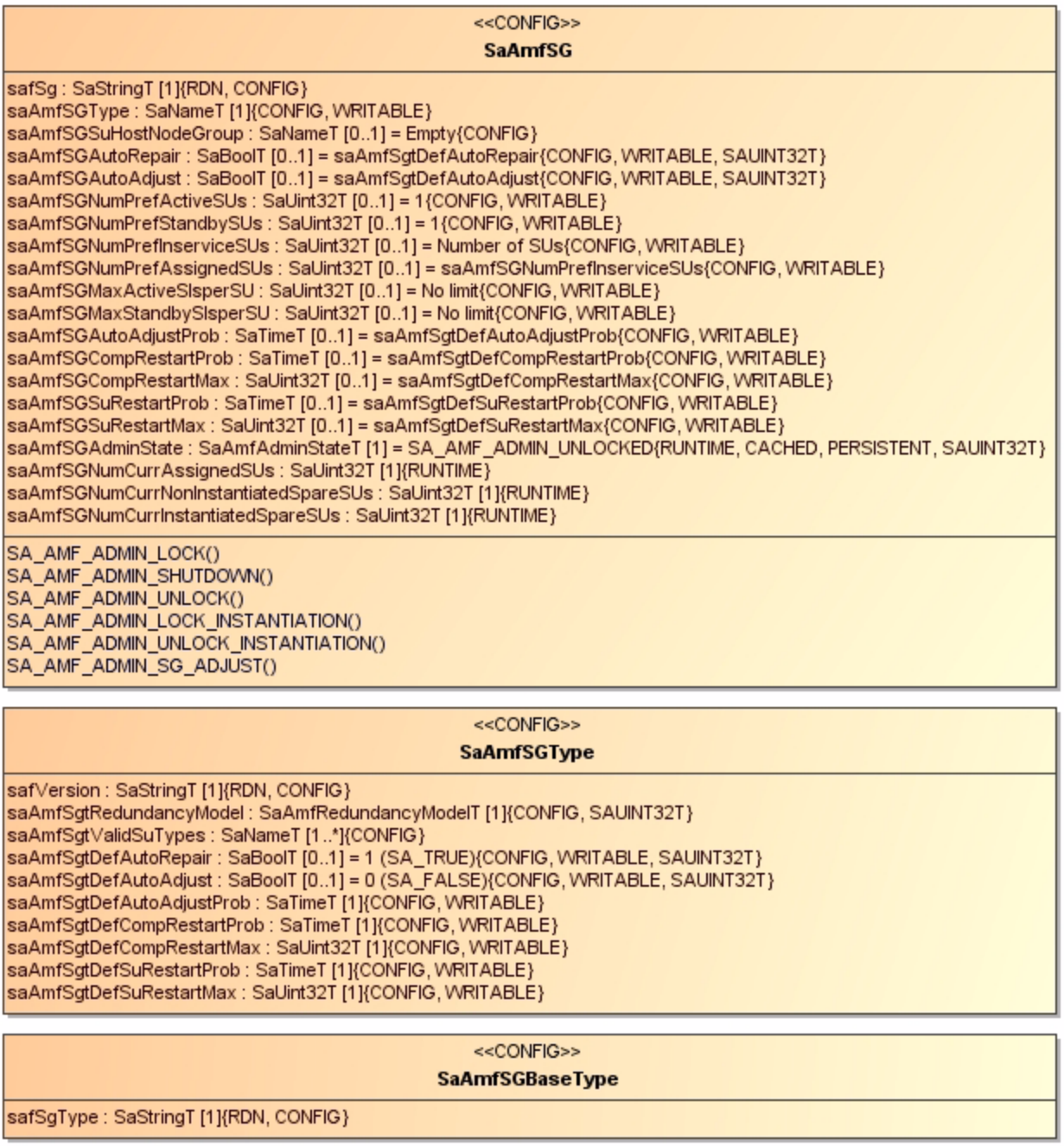


8.9 Service Group Class Diagram

This diagram contains the following classes:

- `SaAmfSG` — This configuration object class defines configuration and runtime attributes of a service group and the operations that can be applied on the service group. For each service group, an object of this class must be configured, and its `saAmfSGType` attribute must contain the DN of a valid object of the `SaAmfSGType` object class. Additional configuration attributes of a service group are defined in the `SaAmfSGType` and the `SaAmfNodeGroup` (see Section 8.7) object classes. For configuring a node group on which local service units can be instantiated, refer to Section 3.1.9 on page 57.
- `SaAmfSGType` — This configuration object class defines configuration attributes of a service group type. The service group type is a generalization of similar service groups that follow the same redundancy model, provide similar availability, and are composed of units of the same service unit types. A service unit type defined in the service group type must be such that any service unit of this service unit type belonging to a service group of a service group type must be capable of supporting a common set of service types. All service groups of the same type share the attribute values defined in the service group type configuration. Some of the attribute values of the service group type may be overridden in the configuration of any service group by setting the corresponding attribute of the configuration object of the service group to the required value.
- `SaAmfSGBaseType` — This configuration object class defines the configuration attributes common to different service group types. In particular, a base service group type defines the common name of versioned service group types. A service group type x belongs to a base service group type y based on the DN of x, which is the concatenation of the RDN of x (representing its version) with the DN of y.

FIGURE 32 3.5- AMF SG Classes



8.10 Service Unit Class Diagram

This diagram contains the following classes:

- `SaAmfSU` — This configuration object class defines configuration and runtime attributes of a service unit and the operations that can be applied on the service unit. For each service unit, an object of this class must be configured, and its `saAmfSUType` attribute must contain the DN of a valid object of the `SaAmfSUType` object class. Additional configuration attributes of a service unit are defined

  ⇒ in the `SaAmfSUType` object class,

  ⇒ in either the `SaAmfNodeGroup` or `SaAmfNode` (see Section 8.7) object classes, and

  ⇒ in the `SaAmfSutCompType` association class.

  For configuring a node or a node group on which a local service unit is instantiated, refer to Section 3.1.9 on page 57.

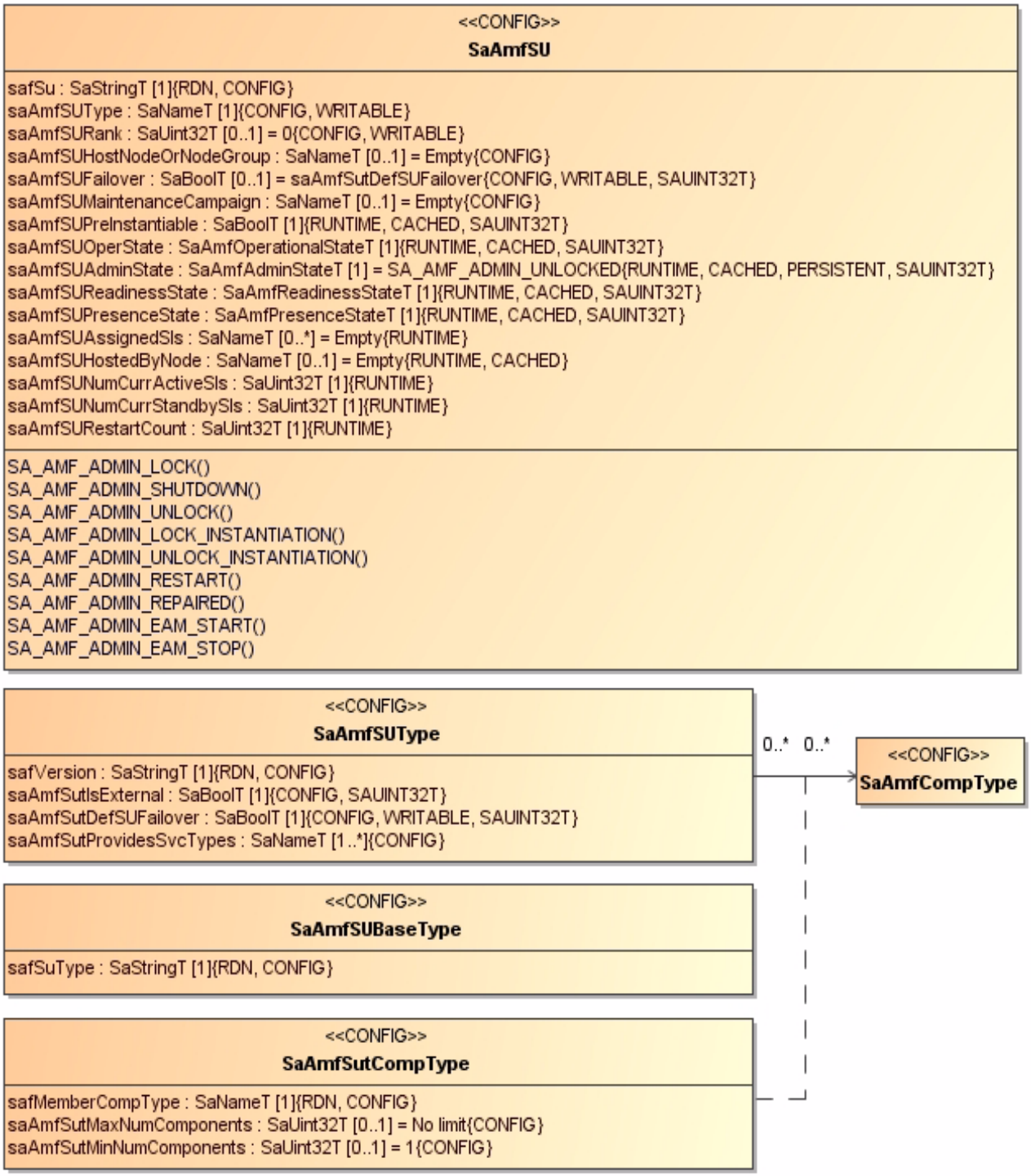
- `SaAmfSUType` — This configuration object class defines configuration attributes of a service unit type. The service unit type defines a list of component types and, for each component type, the number of components that a service unit of this type may accommodate. Each element in this list is expressed by the `SaAmfSutCompType` association class, which is described below. A service unit of a given type may only consist of components of the component types from that list, and the number of these components must be within the range specified for the component type. All service units of the same type share the attribute values defined in the service unit type configuration.
  Some of the attribute values of the service unit type may be overridden in the configuration of any service unit by setting the corresponding attribute of the configuration object of the service unit to the required value. All service units of the same type can be assigned service instances derived from the same set of service types.

- `SaAmfSUBaseType` — This configuration object class defines the configuration attributes common to different service unit types. In particular, a base service unit type defines the common name of versioned service unit types. A service unit type x belongs to a base service unit type y based on the DN of x, which is the concatenation of the RDN of x (representing its version) with the DN of y.

- `SaAmfSutCompType` — This is a configuration association class between the `SaAmfSUType` and `SaAmfCompType` object classes. The `SaAmfSutCompType` class defines configuration attributes of a component type that can be contained in a service unit of the service unit type. An object of this class must be configured for each component type that can be contained in a service unit of the type

defined by the SaAmfSUType object class.
The number of member component types in an service unit type can be determined by the number of saAmfSutCompType objects configured for the service unit type.

FIGURE 33 3.6- AMF SU Classes



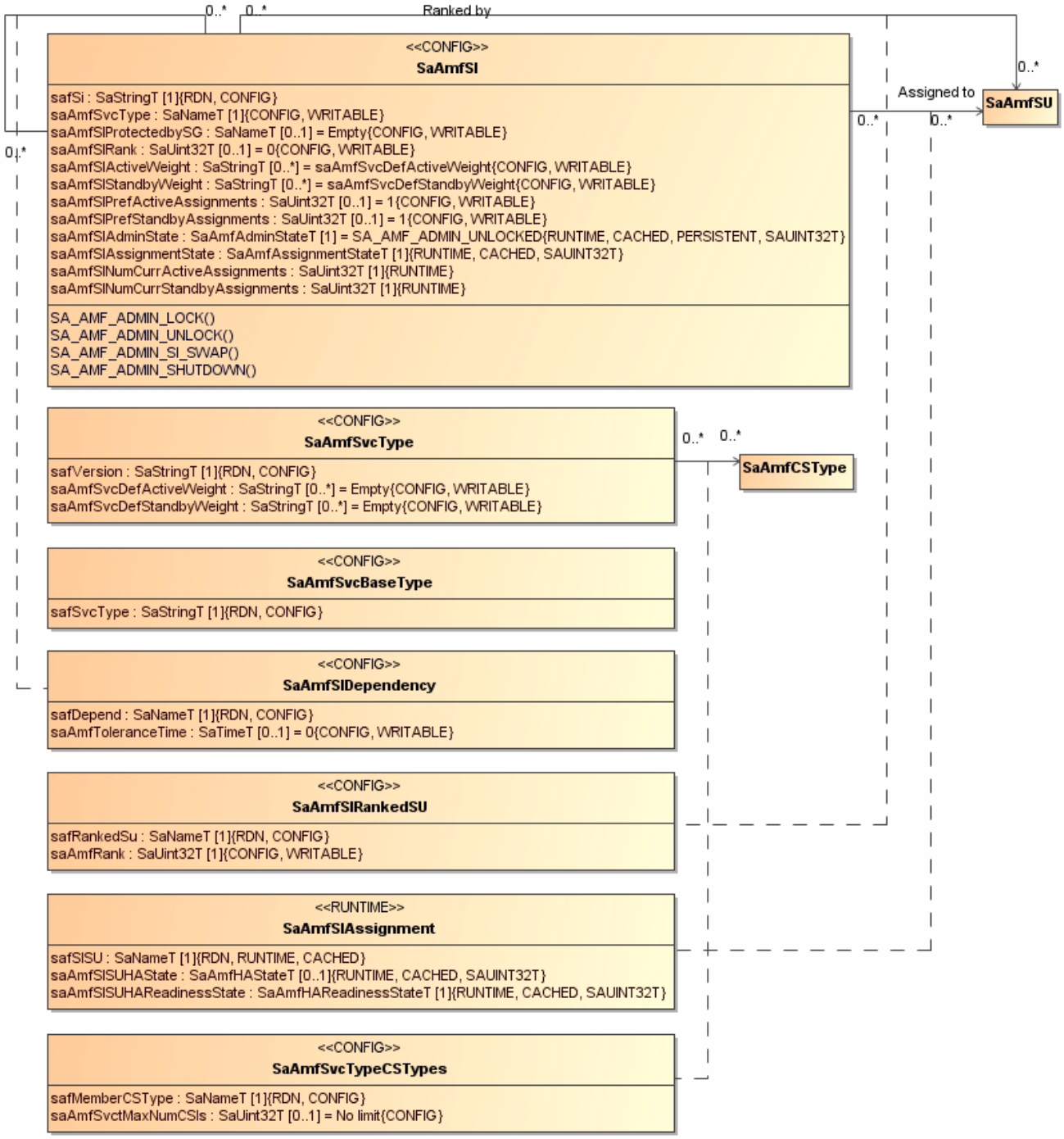
8.11 Service Instance Class Diagram

This diagram contains the following classes:

- SaAmfSI — This configuration object class defines configuration and runtime attributes of a service instance and the operations that can be applied on the service instance. For each service instance, an object of this class must be configured, and its saAmfSvcType attribute must contain the DN of a valid object of the SaAmfSvcType object class. Additional configuration attributes of a service instance are defined in the SaAmfSvcType object class and in the association classes SaAmfSIDependency, SaAmfSIRankedSU, and SaAmfSvcTypeCSTypes.
  The runtime attributes of the assignment of a service instance to a service unit are defined in the SaAmfSIAssignment association class.
- SaAmfSvcType — This configuration object class together with the associated SaAmfSvcTypeCSTypes class defines configuration attributes of a service type. The service type defines a list of component service types of which a service instance may be composed. The service type also defines for each component service type the number of component service instances that a service instance of the given type may aggregate. All service instances of the same type share the attribute values defined in the service type configuration.
- SaAmfSvcBaseType — This configuration object class defines the configuration attributes common to different service types. In particular, a base service type defines the common name of versioned service types. A service type x belongs to a base service type y based on the DN of x, which is the concatenation of the RDN of x (representing its version) with the DN of y.
- SaAmfSIDependency — This is a configuration association class between SaAmfSI object classes. The SaAmfSIDependency class defines configuration attributes for a dependency of a service instance on another service instance, as explained in Section 3.8.1 on page 185. This object class must be configured for each dependency of a service instance on another service instance.
- SaAmfSIRankedSU — This is a configuration association class between the SaAmfSI and the SaAmfSU object classes. The SaAmfSIRankedSU class is used to define the ranked list of service units per service instance, which is required in the N-way (see Section 3.6.4) and N-way active redundancy models (see Section 3.6.5). If an object of this class is not configured, the ranked list of service units for a service instance for the N-way and N-way active redundancy models is given by the ordered list of service units in the service group (this order is configured by setting the saAmfSURank attribute of the SaAmfSU object class, see Section 8.10).

- SaAmfSIAssignment — This is a runtime association class between the SaAmfSI and the SaAmfSU object classes. The SaAmfSIAssignment class defines the attributes of a potential assignment of a service instance to a service unit.
  The saAmfSISUHAReadinessState attribute indicates whether the service instance can be accepted by the service unit, as explained in Section 3.2.1.6, whereas the saAmfSISUHAState attribute, as explained in Section 3.2.1.5, indicates whether the service instance has been assigned to the service unit and in what state.
  An object of this class is first created when either the service instance is assigned to the service unit or a component of the service unit sets its HA readiness state for a component service instance of the particular service instance, and this setting, in turn, changes the HA readiness state of the service unit for the service instance.
- SaAmfSvcTypeCSTypes — This is a configuration association class between the SaAmfSvcType and SaAmfCSType object classes. The SaAmfSvcTypeCSTypes class defines the saAmfSvctMaxNumCSIs configuration attribute to indicate the maximum number of instances of a member CS type (identified by safMemberCSType) that any service instance of a certain service type can have. An object of this class must be configured for each CS type that is contained in a service instance of the type defined by the SaAmfSvcType object class.

FIGURE 34 3.7- AMF SI Classes



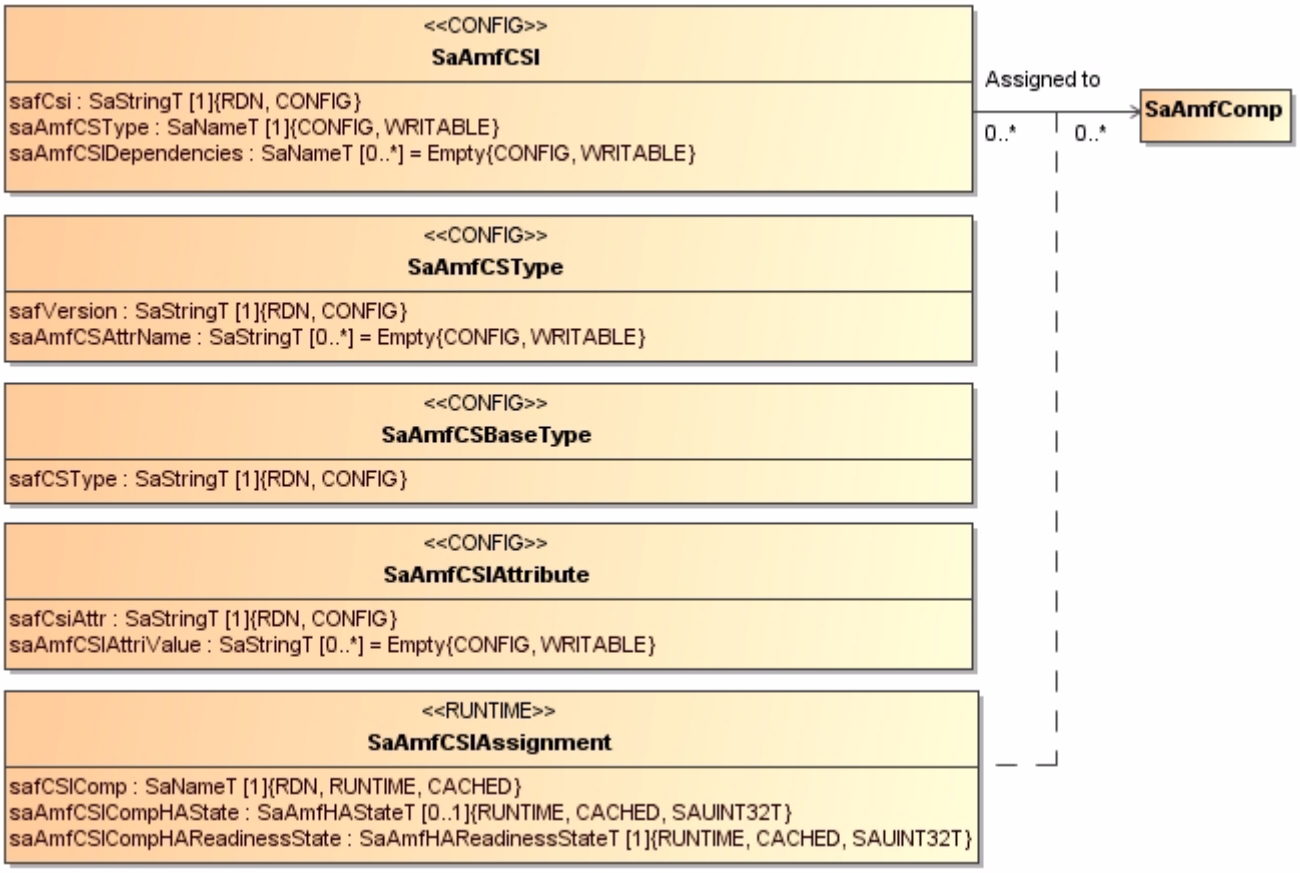
8.12 Component Service Instance Diagram

This diagram contains the following classes:

- `SaAmfCSI` — This configuration object class defines configuration attributes of a component service instance, namely the name of the component service type to which the component service instance belongs and a list of component service instances on which the component service instance depends (see Section 3.8.1.3). For each component service instance, an object of this class must be configured, and its `saAmfCSType` attribute must contain the DN of a valid object of the `SaAmfCSType` object class. Additional configuration attributes of a component service instance are defined in the `SaAmfCSType` and `SaAmfCSIAttribute` object classes.
  The runtime attributes of the assignment of a component service instance to a component are defined in the `SaAmfCSIAssignment` association class.
- `SaAmfCSType` — This configuration object class defines configuration attributes of a component service type. The component service type is the generalization of similar component service instances (that is, similar workloads) that are seen by the Availability Management Framework as equivalent and handled in the same manner. The component service type defines the list of attribute names (as described in Section 3.1.3) for all component service instances belonging to the type.
- `SaAmfCSBaseType` — This configuration object class defines the configuration attributes common to different component service types. In particular, a base component service type defines the common name of versioned component service types. A component service type x belongs to a base component service type y based on the DN of x, which is the concatenation of the RDN of x (representing its version) with the DN of y.
- `SaAmfCSIAttribute` — This configuration object class defines the name and value of an attribute (as described in Section 3.1.3) of a component service instance. An `SaAmfCSIAttribute` object must be defined for each attribute name listed in `SaAmfCSType`.
- `SaAmfCSIAssignment` — This is a runtime association class between the `SaAmfCSI` and the `SaAmfComp` object classes. The `SaAmfCSIAssignment` class defines the attributes of a potential assignment of a component service instance to a component.
  The `saAmfCSICompHAReadinessState` attribute indicates whether the component service instance can be accepted by the component, as explained in Section 3.2.2.5, whereas the `saAmfCSICompHAState` attribute, as explained in Section 3.2.2.4, indicates whether the component service instance has been assigned to the component and in what state. An object of this class is first created when either the component service instance is assigned to the component

or the component sets its HA readiness state for the component service instance.

FIGURE 35 3.8- AMF CSI Classes



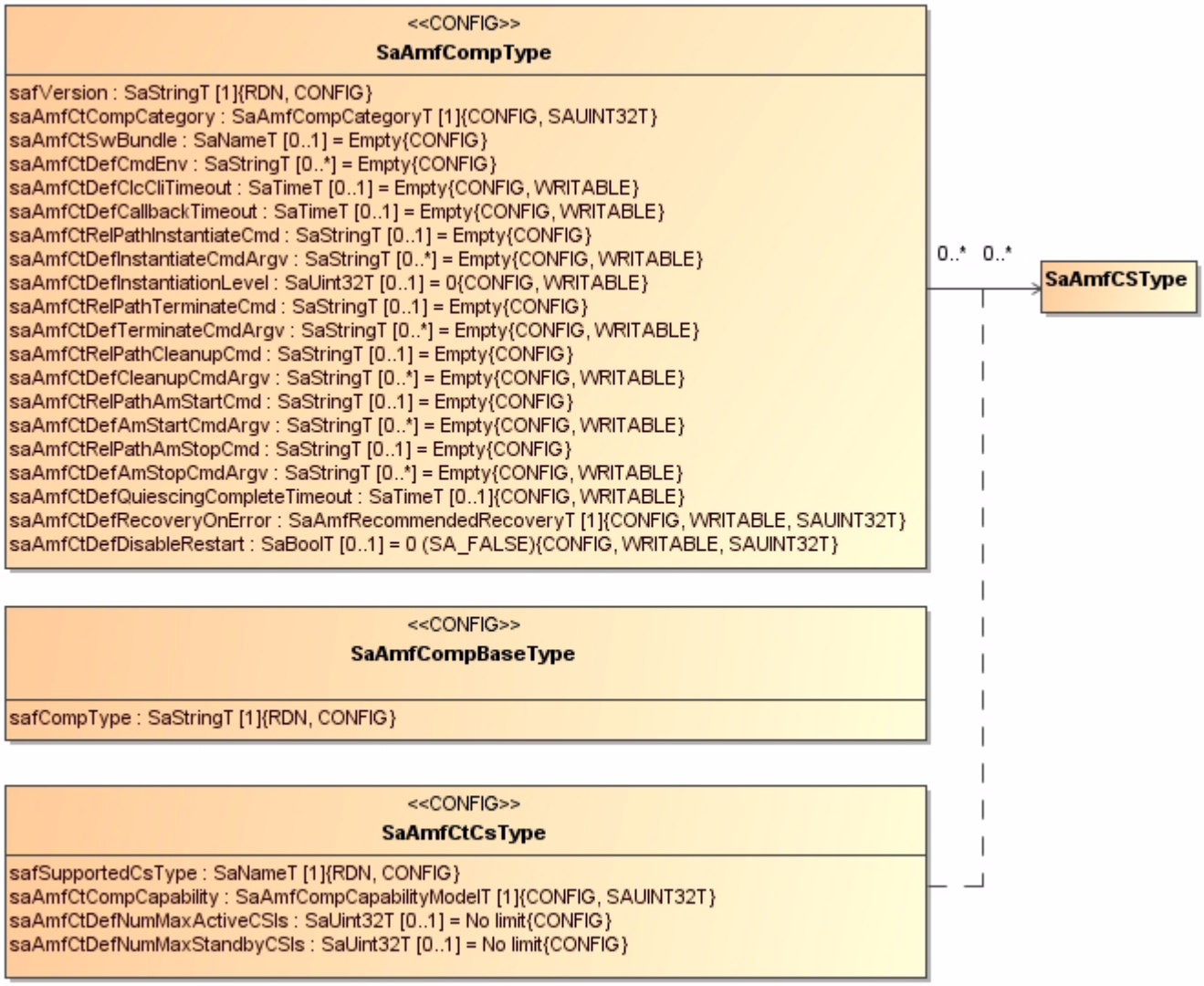
8.13 Component and Component Types Class Diagrams

8.13.1 Component Type Class Diagram

This diagram contains the following classes:

- `SaAmfCompType`—This configuration object class defines configuration attributes of a component type. A component type represents a particular version of the software or hardware implementation that is used to construct components. All components of the same type share the attribute values defined in the component type configuration. Some of the attributes of the component type are defined by the `SaAmfCtCsType` association class, which is described below.
  Some of the attribute values of the component type may be overridden or extended in the configuration of any component by setting the corresponding attribute of the configuration object of the component to the required value.
- `SaAmfCompBaseType`—This configuration object class defines the configuration attributes common to different component types. In particular, a base component type defines the common name of versioned component types. A component type `x` belongs to a base component type `y` based on the DN of `x`, which is the concatenation of the RDN of `x` (representing its version) with the DN of `y`.
- `SaAmfCtCsType`—This is a configuration association class between the `SaAmfCompType` and `SaAmfCSType` object classes. The `SaAmfCtCsType` class defines configuration attributes of a component type for component service instances of a certain component service type (identified by `safSupportedCsType`) that can be assigned to a component of this component type. An object of this class must be configured for each CS type that can be assigned to a component of the type defined by the `SaAmfCompType` object class. For further details, see also the description of the `SaAmfCompCsType` class in Section 8.13.2.
  A component type for a non-pre-instantiable component can only be associated with a single component service type, and the object of class `SaAmfCtCsType` that represents this association must have its `saAmfCtCompCapability` attribute set to `SA_AMF_COMP_NON_PRE_INSTANTIABLE`.
  The `SA_AMF_COMP_NON_PRE_INSTANTIABLE` capability must be set only in association with non-pre-instantiable component types.

FIGURE 36 3.9b- AMF Component Type Classes

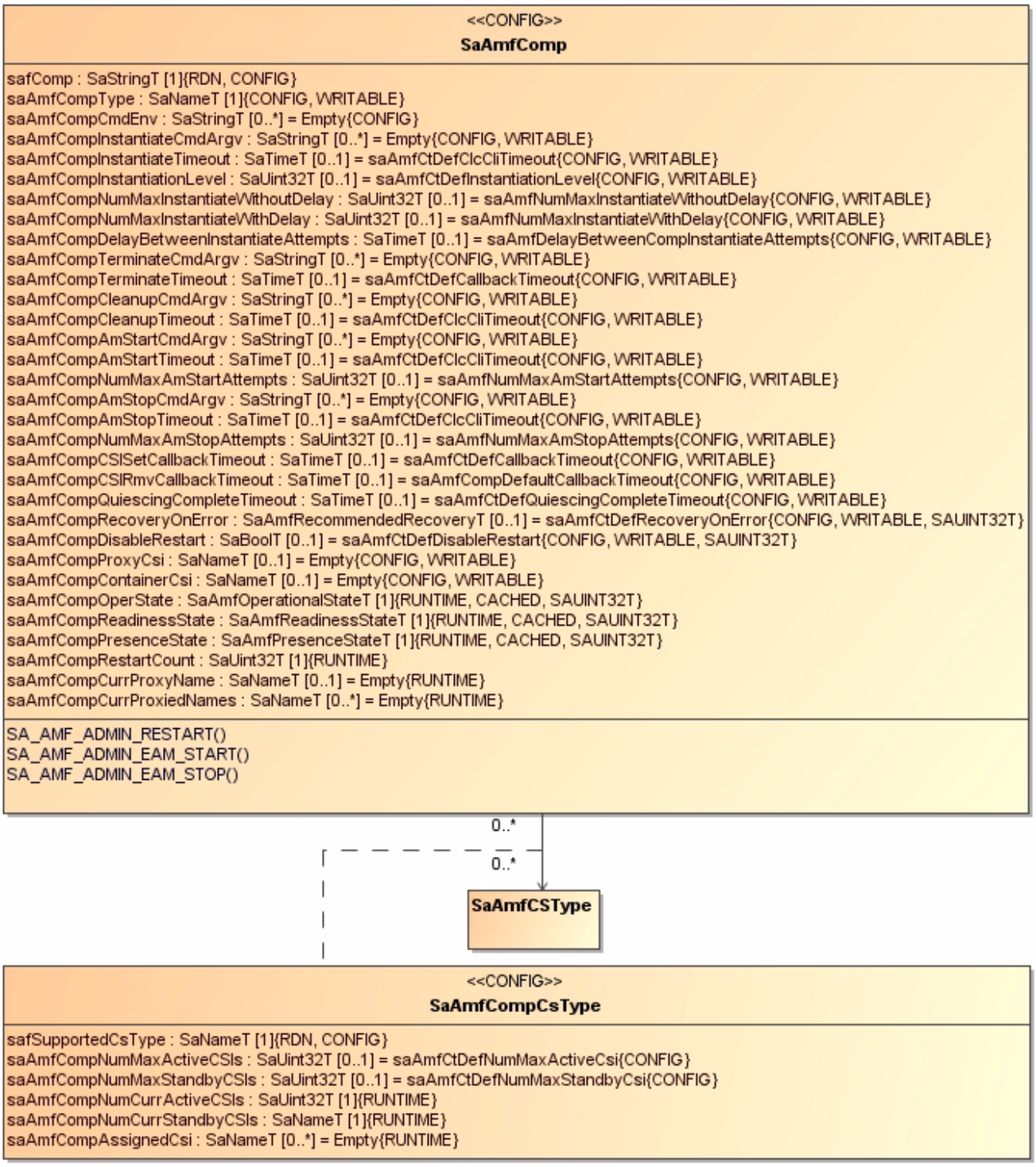


8.13.2 Component Classes Diagram

This diagram contains the following classes:

- `SaAmfComp` — This configuration object class defines configuration and runtime attributes of a component and the operations that can be applied on the component. An object of this class must be configured for each component, and its `saAmfCompType` attribute must contain the DN of a valid object of the `SaAmfCompType` object class. Additional configuration attributes of a component are defined in the `SaAmfCompType` object class and in the `SaAmfCtCsType` association class (see Section 8.13.1), in the `SaAmfCompCSType` association class, and in the `SaAmfCompGlobalAttributes` object class (see Section 8.14).
- `SaAmfCompCsType` — This is a configuration association class between the `SaAmfComp` and `SaAmfCSType` object classes. The `SaAmfCompCsType` class defines configuration and runtime attributes of a component for component service types (each one identified by the attribute `safSupportedCsType`) that can be assigned to the component. An object of this class must be configured for each CS type that can be assigned to a component configured by an object of the `SaAmfComp` object class.
  The attributes of the `SaAmfCompCsType` class are in a particular relation with the attributes of the `SaAmfCtCsType` class. An attribute designated by `saAmfCtDef<attribute name>` in the `SaAmfCtCsType` class defines the default value or an upper limit for the `saAmfComp<attribute name>` attribute of the `SaAmfCompCsType` class. Concerning the rules for overriding or complementing such default values, refer to Section 8.1.

FIGURE 37 3.9a- AMF Component Classes



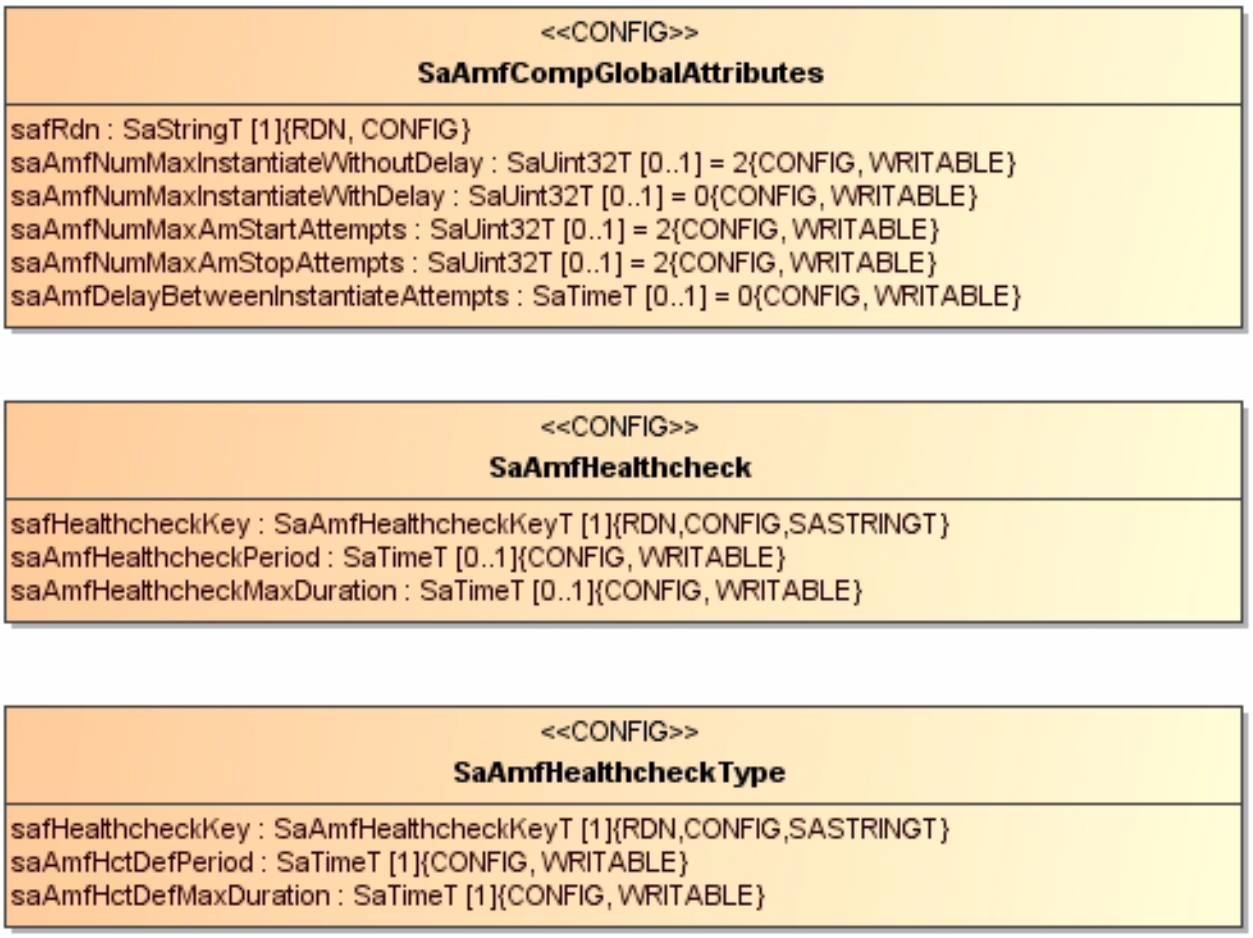
8.14 AMF Global Component Attributes and Healthcheck Classes

This diagram contains the following classes:

- `SaAmfCompGlobalAttributes` — This configuration object class collects those component configuration attributes for which the default value is set globally. Each of these global attributes is referred to by a corresponding attribute in the `SaAmfComp` component configuration object class. One and only one object of this class must be configured for each AMF cluster.
  The global default values can be overridden in the configuration of any component by setting the corresponding attribute of the configuration object of the component to the required value.

  Example: `saAmfNumMaxInstantiateWithoutDelay` in `SaAmfCompGlobalAttributes` corresponds to `saAmfCompNumMaxInstantiateWithoutDelay` in the `SaAmfComp` class.
- `SaAmfHealthcheck` — This configuration object class defines the attributes for a component healthcheck for a certain healthcheck key (see also Section 7.1.2.4). If an object of this class is configured for a component, its attribute values override the corresponding attributes provided in the healthcheck type configuration (see the `SaAmfHealthcheckType` object class) for the component type and for the same healthcheck key. The healthcheck configuration for the component can only specify healthcheck keys for which there is a healthcheck type configuration for its component type.
  The IMM object representing the component healthcheck has a DN of the form "`safHealthcheckKey=...,safComp=...,safSu=...,safSg=...,safApp=...`".
- `SaAmfHealthcheckType` — This configuration object class defines the attributes for a component healthcheck type (see also Section 7.1.2.4). Each healthcheck type is identified by a healthcheck key. An object of this class must be configured for each healthcheck key that a component of a component type uses to start a healthcheck. All components of the same type share the healthcheck attribute values defined in the healthcheck type configuration.
  The IMM object representing the component healthcheck type has a DN of the form "`safHealthcheckKey=...,safVersion=...,safCompBaseType=...`".

FIGURE 38 3.9c- AMF Global Component Attributes and Healthcheck Classes



9 Administration API

This section describes the various administrative API functions that the IMM Service exposes on behalf of the Availability Management Framework to a system administrator. These API functions are described using a 'C' API syntax. The main clients of this administrative API are system management applications and SNMP agents that typically convert system administration commands (invoked from a management station) to the correct administrative API sequence to yield the wanted result that is expected upon execution of the system administration command.

9.1 Availability Management Framework Administration API Model

The Availability Management Framework administrative API functions are applicable to the entities that are controlled by the Availability Management Framework like service units and service instances. Thus, restarting an AMF node by using an Availability Management Framework administration API shall restart all the components contained in the service units housed in the AMF node. This operation will not reboot the underlying CLM node. Similarly, restarting an AMF cluster in the context of Availability Management Framework shall restart all components in the AMF cluster, but shall not reboot the underlying CLM nodes of the CLM cluster.

Most Availability Management Framework administrative API functions are applicable to the service unit (SU) logical entity and entities to which it belongs like a service group (SG), application, AMF node, or AMF cluster. This choice of granularity for administrative operations aligns with Section 3.1.4, which advocates a coarser-grained and aggregated view of components to the system administrator. In certain rare cases, however, the administrative API function directly affects a component within a service unit.

Administrative operations that are applicable to the lowest granular logical entity are called **primitive administrative operations** or simply **primitive operations**. As explained above and in most cases, the lowest granular logical entity is a service unit. The semantics of certain other administrative operations imply a repetitive execution of the same primitive administrative operation to yield the wanted result. These operations are called **composite administrative operations** or simply **composite operations**. For an example, starting external active monitoring (EAM) on a service unit involves starting external active monitoring on all the components housed in the AMF node. Thus, in this case, starting EAM on the service unit is a composite operation, and starting EAM on an individual component is a primitive operation.

In the remainder of this section, it is assumed that concurrent and potentially conflicting administrative operations are invalid, that is, when an administrator has initiated an administrative operation on a logical entity *X*, any other administrative operation that involves a logical entity with which this logical entity *X* has a relationship (association or aggregation) will not be allowed until the first operation on *X* is done.
Note that the shutdown administrative operation is non-blocking, which means that it may complete while the actual procedure to shut down the target entity is still in progress (that is, the entity has not yet reached the locked administrative state). As soon as the shutdown administrative operation completes, other administrative operations such as lock or unlock can be invoked; they can interrupt the shutdown procedure and force the target entity into a locked or an unlocked administrative state.

A general principle that has been adhered to while specifying these administrative operations is that an operation done at a given scope can only be undone by performing the reverse operation at the same scope. This means, for example, one cannot lock at the AMF node-level and then unlock each service unit one by one at the service unit-level. This principle is especially applicable to administrative operations that manipulate the administrative state.

These API functions will be exposed by the IMM Service Object Management library (see [6]).

9.2 Include File and Library Name

The appropriate IMM Service header file and the Availability Management Framework header file must be included in the source of an application using the Availability Management Framework administration API; for the name of the IMM Service header file, see [6]).

To use the Availability Management Framework administration API, an application must be bound to the IMM Service library (for the library name, see [6]).

9.3 Type Definitions

The specification of Availability Management Framework Administration API requires the following types, in addition to the ones already described.

9.3.1 SaAmfAdminOperationIdT

```
typedef enum {
     SA_AMF_ADMIN_UNLOCK                    = 1,
     SA_AMF_ADMIN_LOCK                      = 2,
     SA_AMF_ADMIN_LOCK_INSTANTIATION        = 3,
     SA_AMF_ADMIN_UNLOCK_INSTANTIATION      = 4,
     SA_AMF_ADMIN_SHUTDOWN                  = 5,
     SA_AMF_ADMIN_RESTART                   = 6,
     SA_AMF_ADMIN_SI_SWAP                   = 7,
     SA_AMF_ADMIN_SG_ADJUST                 = 8,
     SA_AMF_ADMIN_REPAIRED                  = 9,
     SA_AMF_ADMIN_EAM_START                 = 10,
     SA_AMF_ADMIN_EAM_STOP                  = 11
} SaAmfAdminOperationIdT;
```

9.4 Availability Management Framework Administration API

As explained earlier, the administrative API shall be exposed by the IMM Service library.

The administrative APIs are described with the assumption that the Availability Management Framework is an object implementer (runtime owner) for the various administrative operations that will be initiated as a consequence of invoking the `saImmOmAdminOperationInvoke_3()` or `saImmOmAdminOperationInvokeAsync_3()` functions (see [6]) with the appropriate `operationId` (described in Section 9.3.1) on the entity designated by the name to which `objectName` points.

The return values explained in the following sections for various administrative operations shall be passed by the `operationReturnValue` parameter, which is provided by the invoker of the `saImmOmAdminOperationInvoke_3()` or `saImmOmAdminOperationInvokeAsync_3()` functions to obtain return codes from the object implementer (Availability Management Framework, in this case).

The operations described in the following subsections are applicable to and have the same effects on both pre-instantiable and non-pre-instantiable service units, unless explicitly stated otherwise.

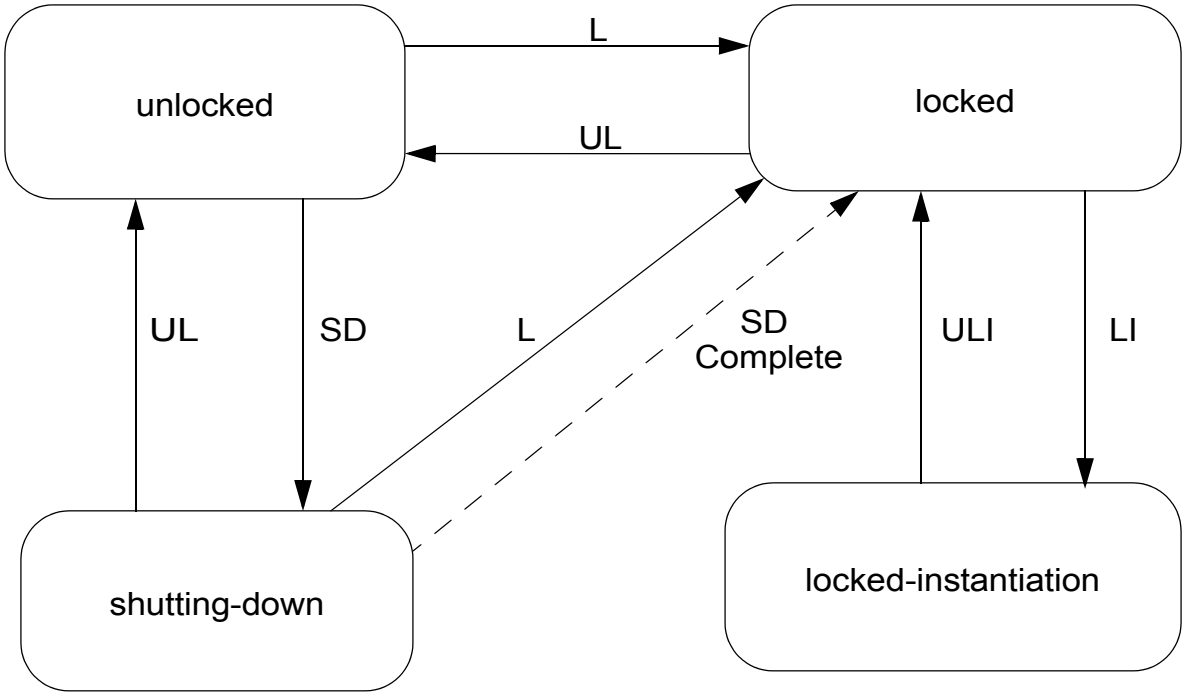
9.4.1 Administrative State Modification Operations

A fair number of administrative operations involve the manipulation of the administrative state. To aid in the description of such administrative operations, FIGURE 39 illustrates the various administrative states and the various operations that are applicable on an entity when it is in a particular administrative state. The abbreviations used in this figure and their meaning are:

- UL = `SA_AMF_ADMIN_UNLOCK`
- L = `SA_AMF_ADMIN_LOCK`
- ULI = `SA_AMF_ADMIN_UNLOCK_INSTANTIATION`
- LI = `SA_AMF_ADMIN_LOCK_INSTANTIATION`
- SD = `SA_AMF_ADMIN_SHUTDOWN`

The dotted line in the figure represents the internal (spontaneous) transition corresponding to the completion of the shutting down operation; this transition moves the entity into locked state without further external intervention.

FIGURE 39 Administrative States and Related Operations for AMF Entities



9.4.2 SA_AMF_ADMIN_UNLOCK

Parameters

`operationId` = `SA_AMF_ADMIN_UNLOCK`

`objectName` - [`in`] A pointer to the name of the logical entity to be unlocked. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to a service unit, a service instance, an AMF node, a service group, an application, and the AMF cluster, that is, to all logical entities that possess an administrative state.

The invocation of this administrative operation transitions the administrative state of the logical entity designated by the name to which `objectName` points to unlocked, provided that the logical entity was previously in the locked or shutting-down administrative state. For more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities, refer to Section 3.2.1.2 on page 63 (service unit), Section 3.2.3.1 on page 87 (service instance), Section 3.2.5 on page 89 (service group), Section 3.2.6.1 on page 90 (AMF node), Section 3.2.7 on page 92 (application), and Section 3.2.8 on page 93 (AMF cluster).

This administrative operation can be issued on a logical entity even if one or more of the AMF nodes hosting the logical entity or parts of it are not mapped to CLM nodes, or one or more of these underlying CLM nodes are not member nodes. It can also be issued on a service unit even if it is configured but uninstantiated.

If this operation is invoked on a logical entity that is already unlocked, there is no change in the status of such an entity, that is, it remains in unlocked state and the caller is returned a benign `SA_AIS_ERR_NO_OP` error code.

If this operation is invoked on a logical entity that is locked for instantiation, there is no change in the status of such an entity, that is, it remains in the locked-instantiation state, and the caller is returned an `SA_AIS_ERR_BAD_OPERATION` error value.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

SA_AIS_ERR_NO_MEMORY - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported by the type of entity denoted by the name to which `objectName` points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of the logical entity, as it is already in unlocked state.

SA_AIS_ERR_BAD_OPERATION - The operation was not successful because the target entity is in locked-instantiation administrative state.

See Also

SA_AMF_ADMIN_LOCK, SA_AMF_ADMIN_SHUTDOWN

9.4.3 SA_AMF_ADMIN_LOCK

Parameters

`operationId` = `SA_AMF_ADMIN_LOCK`

`objectName` - [`in`] A pointer to the name of the logical entity to be locked. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to a service unit, a service instance, an AMF node, a service group, an application, and the AMF cluster, that is, to all logical entities that support an administrative state.

The invocation of this administrative operation transitions the administrative state of the logical entity designated by the name to which `objectName` points to locked, provided that the logical entity was previously in the unlocked or shutting-down administrative state. For more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities, refer to Section 3.2.1.2 on page 63 (service unit), Section 3.2.3.1 on page 87 (service instance), Section 3.2.5 on page 89 (service group), Section 3.2.6.1 on page 90 (AMF node), Section 3.2.7 on page 92 (application), and Section 3.2.8 on page 93 (AMF cluster).

When a service unit or any of the entities containing the service unit is locked, and the service unit contains container components, the Availability Management Framework first performs the following actions for each container component:

- for each associated contained component and for each of its component service instances that has the active HA state and needs to be quiesced, the Availability Management Framework sets the HA state of the associated contained component to quiesced;
- the Availability Management Framework waits for each associated contained component to quiesce for its component service instances (if the setting of the HA state to quiesced was necessary), then it removes all component service instances assigned to the contained component and terminates it (see also page 81).

Analogously, when a service instance containing a container CSI is locked, the Availability Management Framework performs the same actions for contained components whose life cycle is being handled by the associated container component for this container CSI, before it locks the service instance.

This administrative operation can be issued on a logical entity even if one or more of the AMF nodes hosting the logical entity or parts of it are not mapped to CLM nodes, or one or more of these underlying CLM nodes are not member nodes. It can also be issued on a service unit even if it is configured but uninstantiated.

If this operation is invoked by a client on a logical entity that is already locked, there is no change in the status of such an entity, that is, it remains in the locked state, and a benign error value `SA_AIS_ERR_NO_OP` is returned to the client conveying that the entity in question designated by the name to which `objectName` points is already in locked state.

If this operation is invoked on a logical entity that is locked for instantiation, there is no change in the status of such an entity, that is, it remains in the locked-instantiation state, and the caller is returned an `SA_AIS_ERR_BAD_OPERATION` error value.

Chapter 10 provides sequence diagrams to illustrate the actions performed for the lock administrative operation (Section 10.4, Section 10.5, Section 10.8, and Section 10.9).

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

`SA_AIS_ERR_NO_MEMORY` - The Availability Management Framework or a library is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - There are insufficient resources (other than memory).

`SA_AIS_ERR_NOT_SUPPORTED` - This administrative operation is not supported by the type of entity denoted by the name to which `objectName` points.

`SA_AIS_ERR_NO_OP` - The invocation of this administrative operation has no effect on the current state of the logical entity, as it is already in locked state.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not cooperate with the Availability Management Framework in carrying out the administrative operation, the Availability Management Framework tries to terminate them as part of the recovery operation before returning from the operation. If the Availability Management Framework cannot terminate these erroneous components, it will put them in the termination-failed presence state. However, the Availability Management Framework will continue the administrative operation, but will return from the call with this error value, before initiating the required repair operations for such components. The caller of the administrative operation is responsible for discovering such erroneous components and tracking the completion of the subsequent repair operations.

SA_AIS_ERR_BAD_OPERATION - The operation was not successful because the target entity is in locked-instantiation administrative state.

See Also

SA_AMF_ADMIN_UNLOCK

9.4.4 SA_AMF_ADMIN_LOCK_INSTANTIATION

Parameters

`operationId` = `SA_AMF_ADMIN_LOCK_INSTANTIATION`

`objectName` - [`in`] A pointer to the name of the logical entity to be locked for instantiation. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to a service unit, an AMF node, a service group, an application, and the AMF cluster.

The invocation of this administrative operation transitions the administrative state of the logical entity designated by the name to which `objectName` points to locked-instantiation, provided that the logical entity was previously in the locked administrative state. For more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities, refer to Section 3.2.1.2 on page 63 (service unit), Section 3.2.5 on page 89 (service group), Section 3.2.6.1 on page 90 (AMF node), Section 3.2.7 on page 92 (application), and Section 3.2.8 on page 93 (AMF cluster).

After successful invocation of this procedure, all components in all affected service units are terminated and become non-instantiable; in particular, all processes in those components must cease to exist.

The effect of this operation can only be reversed by applying another administrative operation designated by the `operationId` `SA_AMF_ADMIN_UNLOCK_INSTANTIATION`, which causes the relevant service units to be instantiated in a locked state, provided that the entity is not locked for instantiation at any other level, the concerned service units are pre-instantiable, and the redundancy model of the pertinent service groups allows the instantiation. Note that for non-pre-instantiable service units, the application of `SA_AMF_ADMIN_LOCK_INSTANTIATION` is semantically equivalent to the application of `SA_AMF_ADMIN_LOCK` with regards to the presence state of the service units.

This administrative operation can be issued on a logical entity even if one or more of the AMF nodes hosting the logical entity or parts of it are not mapped to CLM nodes, or one or more of these underlying CLM nodes are not member nodes. It can also be issued on a service unit even if it is configured but uninstantiated.

If the logical entity is unavailable during the invocation of this administrative operation, for example, if an AMF node is configured but not a member, all service units within the scope of the entity are set to non-instantiable; they can only ever again be instantiated in a locked state after another administrative operation designated by the `operationId` `SA_AMF_ADMIN_UNLOCK_INSTANTIATION` (refer to Section 9.4.5 on page 378) is invoked on the entity provided that the entity is not locked for instantiation at any other level.

If this operation is invoked by a client on a logical entity that is already in locked-instantiation state, the status of such an entity does not change, that is, the entity remains in that state, and a benign error value `SA_AIS_ERR_NO_OP` is returned to the client, conveying that the state of the concerned entity in question did not change.

If this operation is invoked by a client on a logical entity that is either in the shutting-down or unlocked administrative state, the status of such an entity does not change, that is, the entity remains in the respective state, and the caller is returned an `SA_AIS_ERR_BAD_OPERATION` error value.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

`SA_AIS_ERR_NO_MEMORY` - The Availability Management Framework or a library is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - There are insufficient resources (other than memory).

`SA_AIS_ERR_NOT_SUPPORTED` - This administrative operation is not supported by the type of entity denoted by the name to which `objectName` points.

`SA_AIS_ERR_NO_OP` - The invocation of this administrative operation has no effect on the current state of the logical entity and it remains in the current state.

`SA_AIS_ERR_BAD_OPERATION` - The operation was not successful because the target entity is either in the shutting-down or unlocked administrative state.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not cooperate with the Availability Management Framework in carrying out the administrative operation, the Availability Management Framework tries to terminate them as part of the recovery operation before returning from the operation. If the Availability Management Framework cannot terminate these erroneous components, it will put them in the termination-failed presence state. However, the Availability Management Framework will continue the administrative operation, but will return from the call with this error value, before initiating the required repair operations for such components. The caller of the administrative operation is responsible for discovering such erroneous components and tracking the completion of the subsequent repair operations.

See Also

SA_AMF_ADMIN_UNLOCK_INSTANTIATION, SA_AMF_ADMIN_LOCK

9.4.5 SA_AMF_ADMIN_UNLOCK_INSTANTIATION

Parameters

`operationId` = `SA_AMF_ADMIN_UNLOCK_INSTANTIATION`

`objectName` - [`in`] A pointer to the name of the logical entity to be unlocked for instantiation. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to a service unit, an AMF node, a service group, an application, and the AMF cluster, that is, to all logical entities that support an administrative state with a locked-instantiation value.

The invocation of this administrative operation transitions the administrative state of the logical entity designated by the name to which `objectName` points to locked, provided that the logical entity was previously in the locked-instantiation administrative state. For more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities, refer to Section 3.2.1.2 on page 63 (service unit), Section 3.2.5 on page 89 (service group), Section 3.2.6.1 on page 90 (AMF node), Section 3.2.7 on page 92 (application), and Section 3.2.8 on page 93 (AMF cluster).

If the current administrative state of the target entity is locked-instantiation, the invocation of this operation on such an entity causes all of the relevant service units to become instantiable (though they remain in the locked state), provided that all other conditions are met. A subsequent invocation of the `SA_AMF_ADMIN_UNLOCK` administrative operation would make the relevant service units available for service instance assignments by the Availability Management Framework if all containing entities are also in the unlocked administrative state.

This administrative operation can be issued on a logical entity even if one or more of the AMF nodes hosting the logical entity or parts of it are not mapped to CLM nodes, or one or more of these underlying CLM nodes are not member nodes.

If this operation is invoked by a client on a logical entity that is already locked, the status of such an entity does not change, that is, it remains in the locked state, and a benign error value `SA_AIS_ERR_NO_OP` is returned to the client conveying that the entity (designated by the name to which `objectName` points) is already in locked state.

If this operation is invoked by a client on a logical entity that is either in the shutting-down or unlocked administrative state, the status of such an entity does not change, that is, the entity remains in the respective state, and the caller is returned an SA_AIS_ERR_BAD_OPERATION error value.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

SA_AIS_ERR_NO_MEMORY - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported by the type of entity denoted by the name to which objectName points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of the logical entity, as it is already in locked state.

SA_AIS_ERR_BAD_OPERATION - The operation was not successful because the target entity is either in the shutting-down or unlocked administrative state.

See Also

SA_AMF_ADMIN_LOCK_INSTANTIATION, SA_AMF_ADMIN_UNLOCK

9.4.6 SA_AMF_ADMIN_SHUTDOWN

Parameters

`operationId` = `SA_AMF_ADMIN_SHUTDOWN`

`objectName` - [`in`] A pointer to the name of the logical entity to be shut down. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to a service unit, a service instance, an AMF node, a service group, an application, and the AMF cluster, that is, to all logical entities that support an administrative state.

This administrative operation transitions directly or indirectly the administrative state of the logical entity designated by the name to which `objectName` points to the locked state, provided that the logical entity was previously in the unlocked administrative state.
The indirect transition is performed if the logical entity is actively providing service or being used to provide service (the latter case applies if the logical entity is a service instance). The entity first transitions to the shutting-down state, so that any ongoing service can be completed. When the ongoing service is completed, the entity transitions to the locked state.
For more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities, refer to Section 3.2.1.2 on page 63 (service unit), Section 3.2.3.1 on page 87 (service instance), Section 3.2.5 on page 89 (service group), Section 3.2.6.1 on page 90 (AMF node), Section 3.2.7 on page 92 (application), and Section 3.2.8 on page 93 (AMF cluster).

This administrative operation is non-blocking, that is, it does not wait for the logical entity designated by the name to which `objectName` points to transition to the locked administrative state, which can possibly take a very long time.

When a service unit or any of the entities containing the service unit is shut down, and the service unit contains container components, the Availability Management Framework performs the following actions for each container component, before it shuts down the service unit or any of the containing entities:

- for each associated contained component and for each of its component service instances that has the active HA state and needs to be quiesced, the Availability Management Framework sets the HA state of the associated contained component to quiescing;

- the Availability Management Framework waits for each associated contained component to quiesce for its component service instances (if the setting of the HA state to quiescing was necessary), then it removes all component service instances assigned to the contained component and terminates it (see also page 81).

Analogously, when a service instance containing a container CSI is shut down, the Availability Management Framework performs the same actions for contained components whose life cycle is being handled by the associated container component for this container CSI, before it shuts down the service instance.

If this operation is invoked on a logical entity that is already in shutting-down administrative state, there is no change in the status of such an entity, that is, it continues shutting down, and the caller is returned a benign `SA_AIS_ERR_NO_OP` error value, which means that the entity is already shutting down.

If this operation is invoked by a client on a logical entity that is either in locked or locked-instantiation administrative state, there is no change in the status of such an entity, that is, it remains locked or locked for instantiation, and the caller is returned an `SA_AIS_ERR_BAD_OPERATION` error value.

Chapter 10 provides sequence diagrams to illustrate the actions performed for the shutdown administrative operation (Section 10.1, Section 10.2, Section 10.3, and Section 10.7).

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

`SA_AIS_ERR_NO_MEMORY` - The Availability Management Framework or a library is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` - There are insufficient resources (other than memory).

`SA_AIS_ERR_NOT_SUPPORTED` - This administrative operation is not supported by the type of entity denoted by the name to which `objectName` points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of the logical entity, as it is already in shutting-down state.

SA_AIS_ERR_BAD_OPERATION - The operation was not successful because the target entity is locked or locked for instantiation.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not cooperate with the Availability Management Framework in carrying out the administrative operation, the Availability Management Framework tries to terminate them as part of the recovery operation before returning from the operation. If the Availability Management Framework cannot terminate these erroneous components, it will put them in the termination-failed presence state. However, the Availability Management Framework will continue the administrative operation, but will return from the call with this error value, before initiating the required repair operations for such components. The caller of the administrative operation is responsible for discovering such erroneous components and tracking the completion of the subsequent repair operations.

See Also

SA_AMF_ADMIN_LOCK, SA_AMF_ADMIN_UNLOCK

9.4.7 SA_AMF_ADMIN_RESTART

Parameters

`operationId` = `SA_AMF_ADMIN_RESTART`

`objectName` - [`in`] A pointer to the name of the logical entity to be restarted. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This operation is applicable to a component, a service unit, an AMF node, an application, and the AMF cluster. This procedure typically involves a termination action followed by a subsequent instantiation of either the concerned entity or logical entities that belong to the concerned entity.

This administrative operation is applicable to only those service units whose presence state is instantiated. The invocation of this administrative operation on a service unit causes the service unit to be restarted by restarting all the components within it according to the procedures defined in Section 3.11.1.2 on page 191.

The decision to reassign the assigned service instances to another service unit during this operation should be determined by the Availability Management Framework based on the configured recovery policy of the components that make up the service unit.

If all components within the service unit have a configured recovery policy of restart, that is, the `saAmfCompDisableRestart` configuration attribute of all components is set to `SA_FALSE` (see the `SaAmfComp` object class in Section 8.13.2), it is not necessary to reassign the assigned service instances; however, if at-least one component within the service unit has the `saAmfCompDisableRestart` configuration attribute set to `SA_TRUE`, a reassignment of the service instances assigned to a service unit during its restart (before termination) must be attempted by the Availability Management Framework in course of this administrative action to prevent potential service disruption. In this case, the Availability Management Framework does not set the presence state of the component to restarting and transitions through the individual terminating, terminated, instantiating, instantiated presence states instead.

When this operation is invoked upon a particular instantiated component of a service unit, the other components of the service unit are not affected by this operation, that is, they are not restarted.

If this operation is invoked upon an instantiated container component or upon an instantiated service unit which contains a container component, the Availability Management Framework implicitly restarts all service units that contain contained components having this container component as the associated container component. The procedure regarding reassignment of service instances to these implicitly restarted service units is as explained above when the operation is invoked upon a service unit.

When invoked upon an AMF node, an application, or the AMF cluster, this operation becomes a composite operation that causes a collective restart of all service units residing within the AMF node, application, or the AMF cluster. To execute such a collective restart of all service units in a particular scope, the Availability Management Framework first completely terminates all pertinent service units and does not start to instantiate them until all service units have been terminated. If the target entity is an application or the AMF cluster, the Availability Management Framework does not perform the usual reassignment (in-order to maintain service) of service instances assigned to the various service units during the execution of the termination phase of the restart procedure. Also note that the instantiation phase of such restarts is executed in accordance with the redundancy model configuration for various service groups with no requirement to preserve pre-restart service instance assignments to various service units in the application or AMF cluster.

The Availability Management Framework must not proceed with this operation if another administrative operation or an error recovery initiated by the Availability Management Framework is already engaged on the logical entity. In such case, the `SA_AIS_ERR_TRY_AGAIN` error value shall be returned to indicate that the action is feasible but not at this instant.

Section 10.10 provides a sequence diagram to illustrate the actions performed for the restart of a container component.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

`SA_AIS_ERR_NO_MEMORY` - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_BAD_OPERATION - The target logical entity for this operation identified by name to which objectName points could not be restarted for various reasons like the presence state of the service unit or the component to be restarted was not instantiated.

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported by the type of entity denoted by the name to which objectName points.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not cooperate with the Availability Management Framework in carrying out the administrative operation, the Availability Management Framework tries to terminate them as part of the recovery operation before returning from the operation. If the Availability Management Framework cannot terminate or instantiate these erroneous components, it will put them in the termination-failed or instantiation-failed presence state. However, the Availability Management Framework will continue the administrative operation, but will return from the call with this error value, before initiating the required repair operations for such components. The caller of the administrative operation is responsible for discovering such erroneous components and tracking the completion of the subsequent repair operations.

See Also

None

9.4.8 SA_AMF_ADMIN_SI_SWAP

Parameters

`operationId` = `SA_AMF_ADMIN_SI_SWAP`

`objectName` - [`in`] A pointer to the name of the service instance whose component service instances need to be swapped. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to service instances that are currently assigned to service units.

The invocation of this procedure results in swapping the HA states of the appropriate CSIs contained within an SI. The typical outcome of this operation results in the HA state of CSIs assigned to components within the service units to be interchanged; active assignments become standby, and standby assignments become active.

The SI identified by the name to which `objectName` points is called here the designated SI.

If the designated SI is protected by a service group whose redundancy model is 2N, the invocation of this administrative operation causes a complete swap of all active and standby CSIs belonging to not just the designated SI but also to any other SI that is assigned to a service unit to which the designated SI is assigned. Note that this behavior is consistent with the semantics of the respective redundancy model.

If the designated SI is protected by a service group whose redundancy model is N+M, the invocation of this administrative operation results in a complete swap of all active and standby CSIs belonging to not just the designated SI but also to any other SI that is assigned active to a service unit to which the designated SI is assigned active. Application of this operation on an SI may potentially modify the standby assignments of other SIs that are protected by the same service group, but are not assigned to the service unit to which the SI in question is assigned active. For an example, refer to FIGURE 15 on page 134: if the swap operation is applied on SI *A*, the active assignment for SI *A* shall be moved to Service Unit *S4* on Node *X*, and the standby assignments for SI *A* as well as that of SI *C* and SI *B* will be moved to Service Unit *S1* on Node *U*. The active assignments of SI *C* and SI *B* will remain on Service Unit *S3* (on Node *W*) and Service Unit *S2* (on Node *V*), respectively.

If the redundancy model of the protecting service group is N-way, the aggregate effect of swapping all SIs assigned to a service unit by swapping only one SI is not achieved. This behavior is again consistent with the semantics of the N-way redun-

dancy model. In the N-way redundancy model, it is possible that an SI has multiple standby assignments, in which case this administrative operation shall affect only the highest-ranked standby assignment.

This operation must not be invoked on an SI that is protected by a service group whose redundancy model is either N-way active or no-redundancy.

If no standby assignments are available for an SI (potentially because the AMF cluster is in a degenerated status, and reduction procedures have been engaged) when this operation is invoked on a particular logical entity, the `SA_AIS_ERR_BAD_OPERATION` error value shall be returned.
In other words, this operation shall be allowed by the Availability Management Framework to proceed under the following circumstances.

- The concerned SI is assigned active or quiescing to one service unit.
- The concerned SI is assigned standby to at least another service unit.
- The HA readiness state of the service unit currently assigned the standby HA state for the target service instance is ready-for-assignment.
- The node capacity limits for the affected AMF nodes are not violated when the HA state assignments are swapped.

The Availability Management Framework shall not proceed with this procedure when the presence state of the constituent service units of the service group protecting the SI is instantiating, restarting, or terminating, and should return an `SA_AIS_ERR_TRY_AGAIN` error value conveying that the action is valid but not currently possible.

The SI-SI dependency rules and dependencies among the component service instances of the same SI must be honored, if applicable during the execution of this operation.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

SA_AIS_ERR_NO_MEMORY - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_BAD_OPERATION - The operation was not successful on the target SI, possibly because one or more of the following conditions are not met:

- The concerned SI is assigned active or quiescing to one service unit.
- The concerned SI is assigned standby to at least another service unit.
- The HA readiness state of the service unit currently assigned the standby HA state for the target service instance is ready-for-assignment.
- The node capacity limits for the affected AMF nodes are not violated when the HA state assignments are swapped.

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported for the type of entity denoted by the name to which `objectName` points.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not cooperate with the Availability Management Framework in carrying out the administrative operation, the Availability Management Framework tries to terminate them as part of the recovery operation before returning from the operation. If the Availability Management Framework cannot terminate these erroneous components, it will put them in the termination-failed presence state. However, the Availability Management Framework will continue the administrative operation, but will return from the call with this error value, before initiating the required repair operations for such components. The caller of the administrative operation is responsible for discovering such erroneous components and tracking the completion of the subsequent repair operations.

See Also

None

9.4.9 SA_AMF_ADMIN_SG_ADJUST

Parameters

`operationId` = `SA_AMF_ADMIN_SG_ADJUST`

`objectName` - [`in`] A pointer to the name of the service group that needs to be transitioned to the original 'preferred configuration'. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This operation is only applicable to a service group.

This operation moves a service group to the preferred configuration, which typically causes the service instance assignments of the service units in the service group to be transferred back to the most preferred service instance assignments in which the highest-ranked available service units are assigned the active or standby HA states for those service instances. If the most preferred configuration cannot be achieved, this operation will restore the best possible configuration in which the rankings of the service units are respected with regards to active and standby service instance assignments.

The objective of this administrative operation is to provide an administrator the capability to manually execute an adjust procedure, as described in Section 3.6.1.1 on page 110. This operation is generally issued after the service group has undergone a series of swaps, locks, or shutdowns, and the invocation of this administrative operation brings the service group back to its initial preferred state or as close to the preferred state as possible.

The Availability Management Framework shall not proceed with this procedure when the presence state of the constituent service units of the service group is instantiating, restarting, terminating, or the administrative state is shutting-down. For these cases, the Availability Management Framework should return an `SA_AIS_ERR_TRY_AGAIN` error value conveying that the action is valid but not currently possible.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

SA_AIS_ERR_NO_MEMORY - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported for the type of entity denoted by the name to which `objectName` points.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not cooperate with the Availability Management Framework in carrying out the administrative operation, the Availability Management Framework tries to terminate them as part of the recovery operation before returning from the operation. If the Availability Management Framework cannot terminate these erroneous components, it will put them in the termination-failed presence state. However, the Availability Management Framework will continue the administrative operation, but will return from the call with this error value, before initiating the required repair operations for such components. The caller of the administrative operation is responsible for discovering such erroneous components and tracking the completion of the subsequent repair operations.

See Also

None

9.4.10 SA_AMF_ADMIN_REPAIRED

Parameters

`operationId` = `SA_AMF_ADMIN_REPAIRED`

`objectName` - [`in`] A pointer to the name of the logical entity to be repaired. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation is applicable to a service unit and an to AMF node.

This administrative operation is used to clear the disabled operational state of an AMF node or a service unit after they have been successfully mended to declare them as repaired. The administrator uses this operation to indicate the availability of a service unit or an AMF node for providing service after an externally executed repair action. When invoked on an AMF node, this operation results in enabling the operational state of the constituent service units and components. When invoked on a service unit, it has similar effect on all the components that make up the service unit. An AMF node or a service unit enters the disabled operational state due to reasons stated in Section 3.2.6.2 on page 91 (AMF node) and Section 3.2.1.3 on page 63 (service unit).

The Availability Management Framework might optionally engage in repairing an AMF node or a service unit after a successful recovery procedure execution, in which case the Availability Management Framework itself will clear the disabled state of the involved AMF node or service unit. However, if the repair action is undertaken by an external entity outside the scope of the Availability Management Framework, or the Availability Management Framework failed to successfully repair (and the repair requires intervention by an external entity), one should use this administrative operation to clear the disabled state of the AMF node or the service unit to indicate that these entities are repaired and their operational state is enabled.

It is expected that a repair done by an external entity should bring the repaired service units and components in a consistent state, that is, to the uninstantiated presence state, before an `SA_AIS_OK` status is returned by this operation.

This administrative operation can be issued on an AMF node or on a service unit hosted by an AMF node even if this AMF node is not mapped to a CLM node, or the underlying CLM node is not a member node. It can also be issued on a service unit even if it is configured but uninstantiated.

If this administrative operation is invoked on an AMF node or a service unit whose operational state is already enabled, the entity remains in that state, and a benign error value of SA_AIS_ERR_NO_OP is returned to the caller.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

SA_AIS_ERR_NO_MEMORY - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported by the type of entity denoted by the name to which objectName points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of the logical entity, as it is already enabled.

SA_AIS_ERR_BAD_OPERATION - The operation could not ensure that the presence states of the relevant service units and components are either instantiated or uninstantiated.

See Also

None

9.4.11 SA_AMF_ADMIN_EAM_START

Parameters

`operationId` = `SA_AMF_ADMIN_EAM_START`

`objectName` - [`in`] A pointer to the name of the logical entity on which external active monitoring needs to be started. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation applies to a component and a service unit.

This API function is invoked to resume external active monitoring of components after it has been stopped by invoking the administrative operation designated by `operationId` = `SA_AMF_ADMIN_EAM_STOP` on the same component.

If a component on which this administrative operation is invoked is already being actively monitored, there is no change in its status as a consequence of invoking this operation on such a component. A status of `SA_AIS_ERR_NO_OP` is returned in such a case.

When this procedure is applied to a service unit, it results in an aggregate action of starting the external active monitors for all components within the service unit that support external active monitoring without affecting the ones that are already being actively monitored. If the external monitors for all components within the enclosing service unit that support external active monitoring have been already started, an `SA_AIS_ERR_NO_OP` error code is returned to indicate that there has been no change in the status of active monitoring of the components within the service unit.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

`SA_AIS_ERR_NO_MEMORY` - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_FAILED_OPERATION - The AM_START operation returns an error or fails to complete within the configured timeout.

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported for the type of entity denoted by the name to which `objectName` points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of active monitoring of the logical entity.

See Also

SA_AMF_ADMIN_EAM_STOP

9.4.12 SA_AMF_ADMIN_EAM_STOP

Parameters

`operationId` = `SA_AMF_ADMIN_EAM_STOP`

`objectName` - [`in`] A pointer to the name of the logical entity on which external active monitoring needs to be stopped. The name is expressed as a LDAP DN. The type of the logical entity is inferred by parsing this DN.

Description

This administrative operation applies to a component and a service unit.

This API function is typically invoked to stop external active monitoring of components before terminating them.

If a component on which this administrative operation is invoked is not being actively monitored, there is no change in its status as a consequence of invoking this operation on such a component. A status of `SA_AIS_ERR_NO_OP` is returned in such a case.

When this procedure is applied to a service unit, it results in an aggregate action of stopping the external active monitors for all components within the service unit that support external active monitoring without affecting the ones that are not being actively monitored. If the external monitors for all components within the enclosing service unit that support external active monitoring have been already stopped, an `SA_AIS_ERR_NO_OP` error code is returned to indicate that there has been no change in the status of active monitoring of the components within the service unit.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error generally should be returned when the requested action is valid but not currently possible, probably because another operation is acting upon the logical entity on which the administrative operation is invoked. Such an operation can be another administrative operation or an error recovery initiated by the Availability Management Framework.

SA_AIS_ERR_NO_MEMORY - The Availability Management Framework or a library is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_FAILED_OPERATION - The AM_STOP operation returns an error or fails to complete within the configured timeout.

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported for the type of entity denoted by the name to which `objectName` points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of active monitoring of the logical entity.

See Also

SA_AMF_ADMIN_EAM_START

9.5 Summary of Administrative Operation Support

The following table summarizes the various administrative operations supported by the various logical entities within the Availability Management Framework system model.

Table 22 Summary: Applicability of Administrative Operations

Administrative Operation	Applicability
UNLOCK	AMF cluster, application, SG, AMF node, SU, SI
LOCK	AMF cluster, application, SG, AMF node, SU, SI
UNLOCK_INSTANTIATION	AMF cluster, application, SG, AMF node, SU
LOCK_INSTANTIATION	AMF cluster, application, SG, AMF node, SU
SHUTDOWN	AMF cluster, application, SG, AMF node, SU, SI
RESTART	AMF cluster, application, AMF node, SU, component
SWAP_SI	SI
ADJUST_SG	SG
REPAIRED	AMF node, SU
EAM_START	SU, component
EAM_STOP	SU, component

10 Basic Operational Scenarios

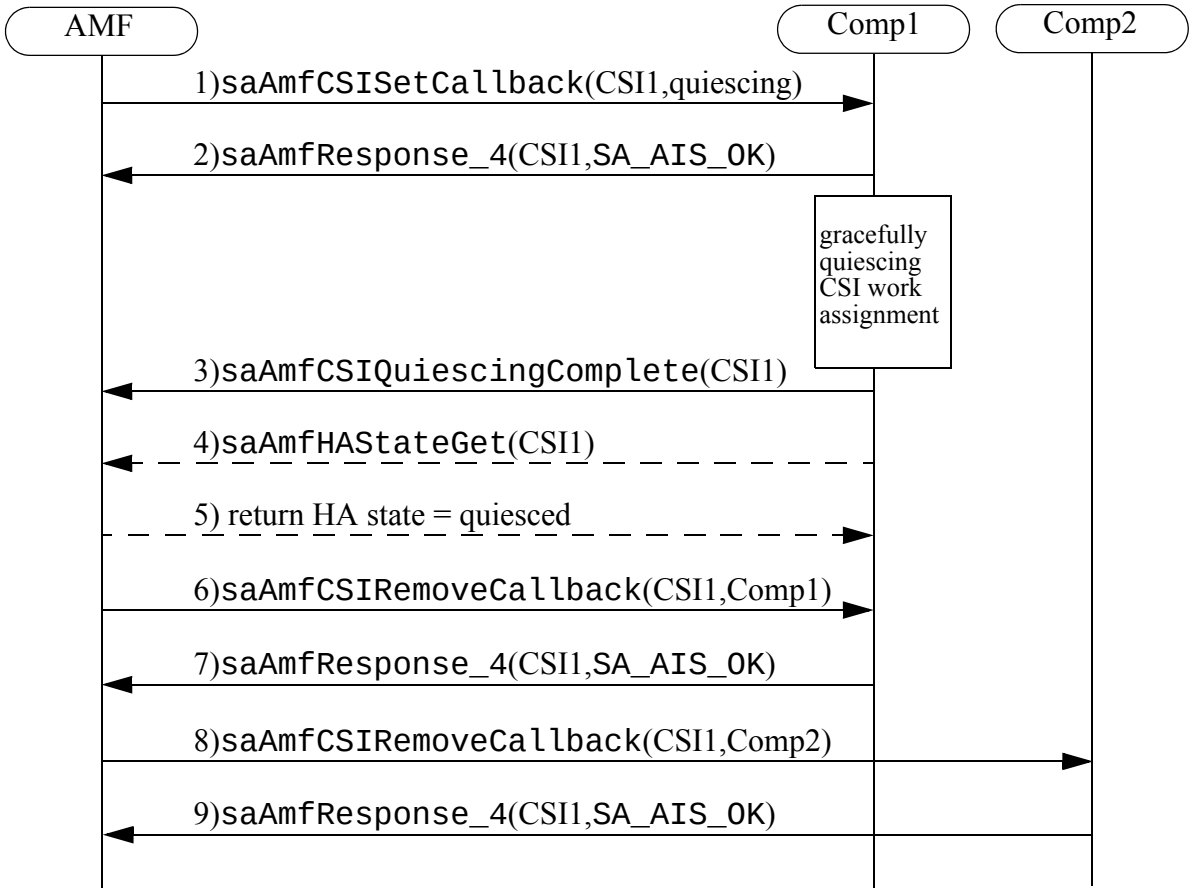
This section contains basic operational scenarios.
For the portrayed operational scenarios, the following assumptions have been made:

- The HA readiness state for all service units and components is ready-for-assignment for all current and future SI and CSI assignments.
- None of the SI assignments made within the operational scenarios would violate the capacity limits of the host nodes.

10.1 Administrative Shutdown of a Service Instance in a 2N Case

The context of this scenario is a service group with 2N redundancy model having two service units with a single regular SA-aware, contained, or proxy component each. Two SIs are assigned to the service unit such that component Comp1 and component Comp2 have each two component service instance assignments: Comp1 is assigned active for CSI1 and CSI2, and Comp2 is assigned standby for CSI1 and CSI2. The following diagram shows the sequence of actions when one of the two SIs is administratively shut down.

FIGURE 40 Administrative Shutdown of a Service Instance for the 2N Case



The dotted lines indicate optional transactions.

The result of a “complete” transition from the quiescing HA state is to arrive at the quiesced HA state.

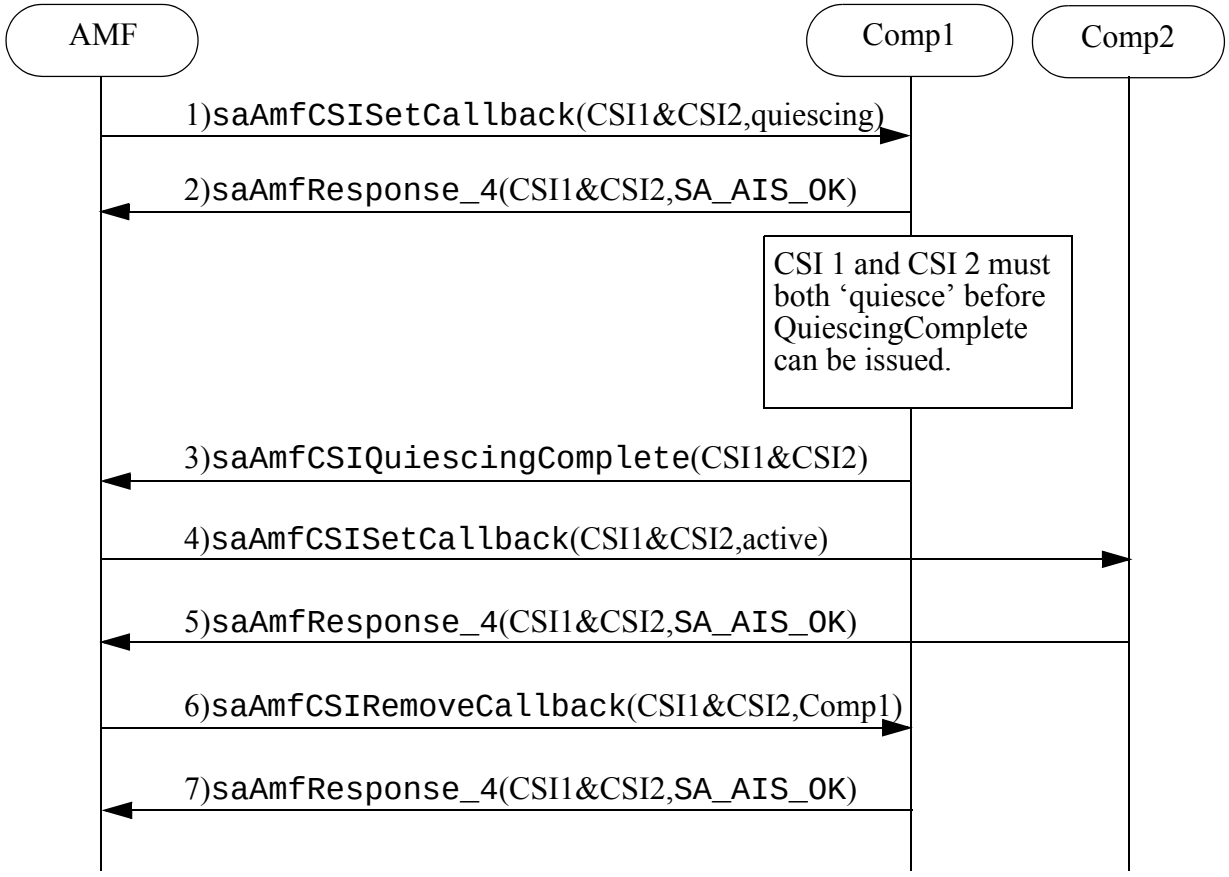
Notice that as only one of the SIs has been shut down, the component service instance corresponding to that SI (CSI1) is manipulated and the other (CSI2) is left unchanged.

Further notice that the Availability Management Framework does not remove the standby state for CSI1 from Comp2 until the active HA state of Comp1 for CSI1 has transitioned successfully to quiesced. At this time, the Availability Management Framework can remove the CSI1 assignment from Comp1 and Comp2 in any order.

10.2 Administrative Shutdown of a Service Unit in a 2N Case

The context of this scenario is a service group with 2N redundancy model having two service units with a single regular SA-aware, contained, or proxy component each. Two SIs are assigned to the service unit such that component Comp1 and component Comp2 have each two component service instance assignments: Comp1 is assigned active for CSI1 and CSI2, and Comp2 is assigned standby for CSI1 and CSI2. The following diagram shows the sequence of actions when one of the service units (the one having components assigned active for CSI1 and CSI2) is administratively shut down.

FIGURE 41 Administrative Shutdown of a Service Unit for the 2N Case



The Availability Management Framework should use the `csiFlags` value `SA_AMF_TARGET_ALL` in the callback of step 4 in order to guarantee that 2N semantics are honored. Those semantics are "...at most one service unit will have the active HA state for all service instances, and at most one service unit will have the standby HA state for all service instances".

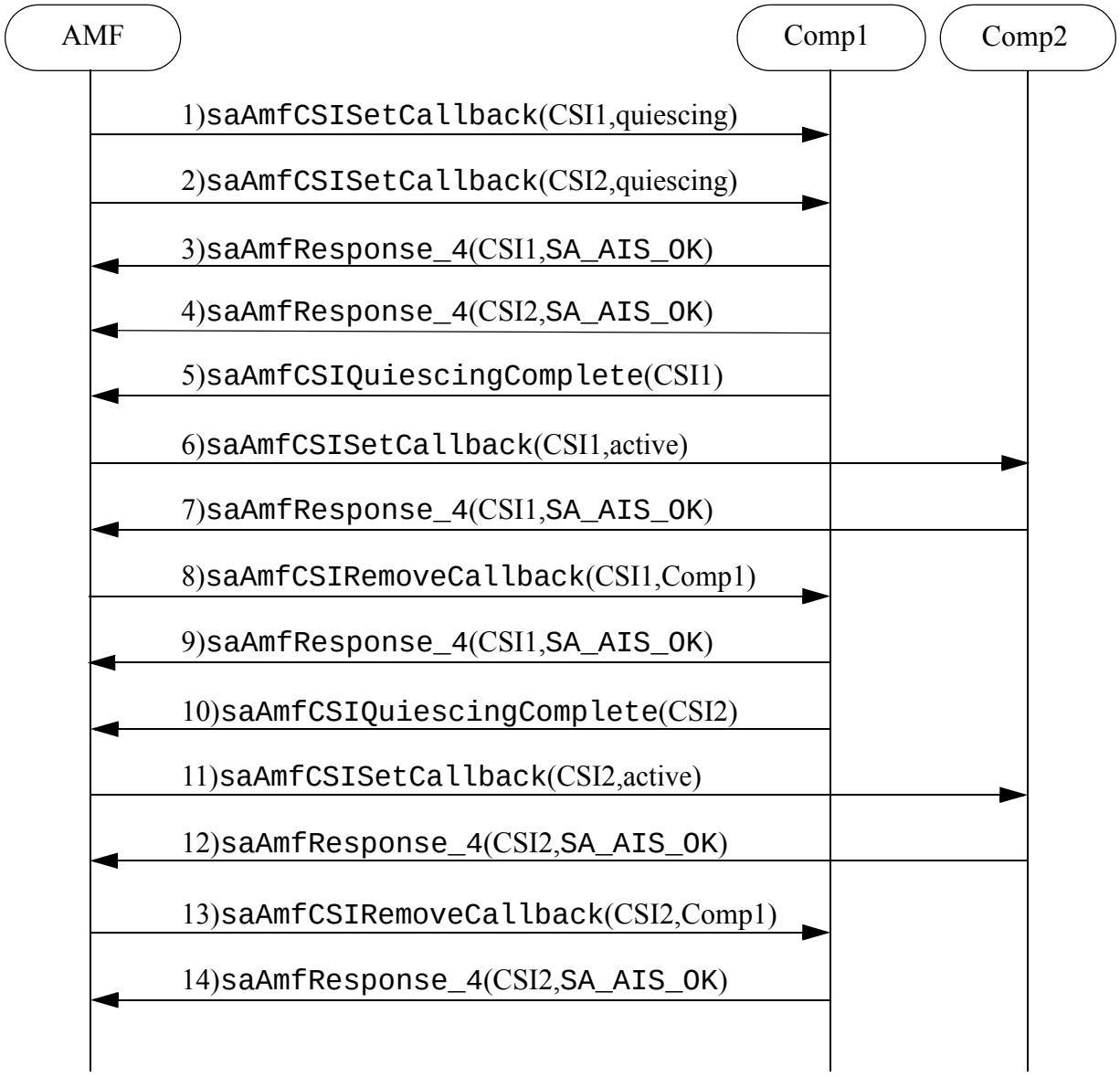
Notice that `saAmfCSIQuiescingComplete()` can only be invoked when all component service instance assignments have successfully quiesced within the component.

10.3 Administrative Shutdown of a Service Unit for the N-Way Model

This scenario is the same as in the previous section, except that the redundancy model is another one.

The context of this scenario is a service group with N-way redundancy model having two service units with a single regular SA-aware, contained, or proxy component each. Two SIs are assigned to the service units such that component Comp1 and component Comp2 have each two CSI assignments: Comp1 is assigned active for CSI1 and CSI2, and Comp2 is assigned standby for CSI1 and CSI2. The following diagram shows the sequence of actions when one of the service units (the one having components assigned active for CSI1 and CSI2) is administratively shut down.

FIGURE 42 Administrative Shutdown of a Service Unit for the N-Way Case



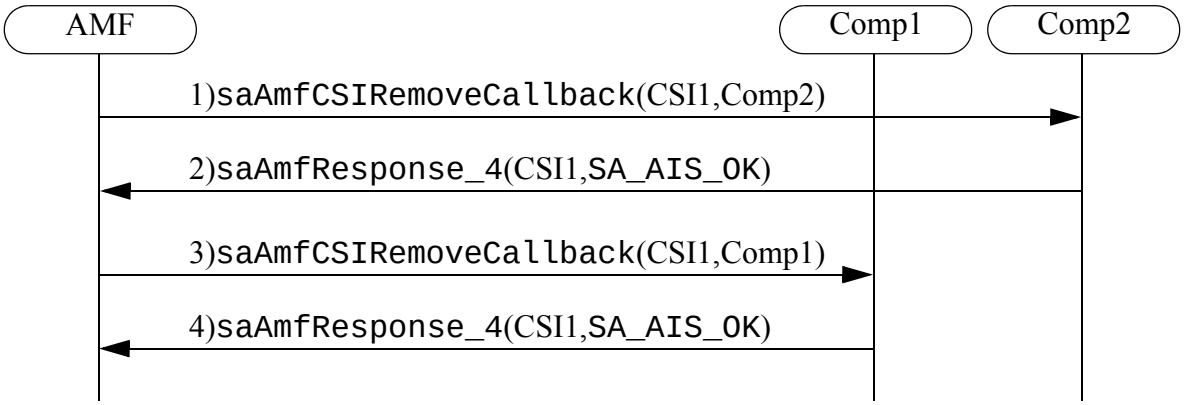
Note that Comp2 will have both active and standby assignments for a certain period of time, which implies that Comp2 must have the x_active_and_y_standby capability.

Also notice that CSI2 at Comp1 has taken much longer to quiesce (from step 2 to step 10) while CSI1 at Comp1 quiesced much faster (from step 1 to step 5) allowing the Availability Management Framework to proceed with the active HA assignment for CSI1 to Comp2.

10.4 Administrative Lock of a Service Instance

The context of this scenario is a service group with 2N redundancy model having two service units with a single regular SA-aware, contained, or proxy component each. Two SIs are assigned to the service units such that component Comp1 and component Comp2 have each two CSI assignments: Comp1 is assigned active for CSI1 and CSI2, and Comp2 is assigned standby for CSI1 and CSI2. The following diagram shows the sequence of actions when one of the two SIs are locked.

FIGURE 43 Administrative Lock of a Service Instance for the 2N Case

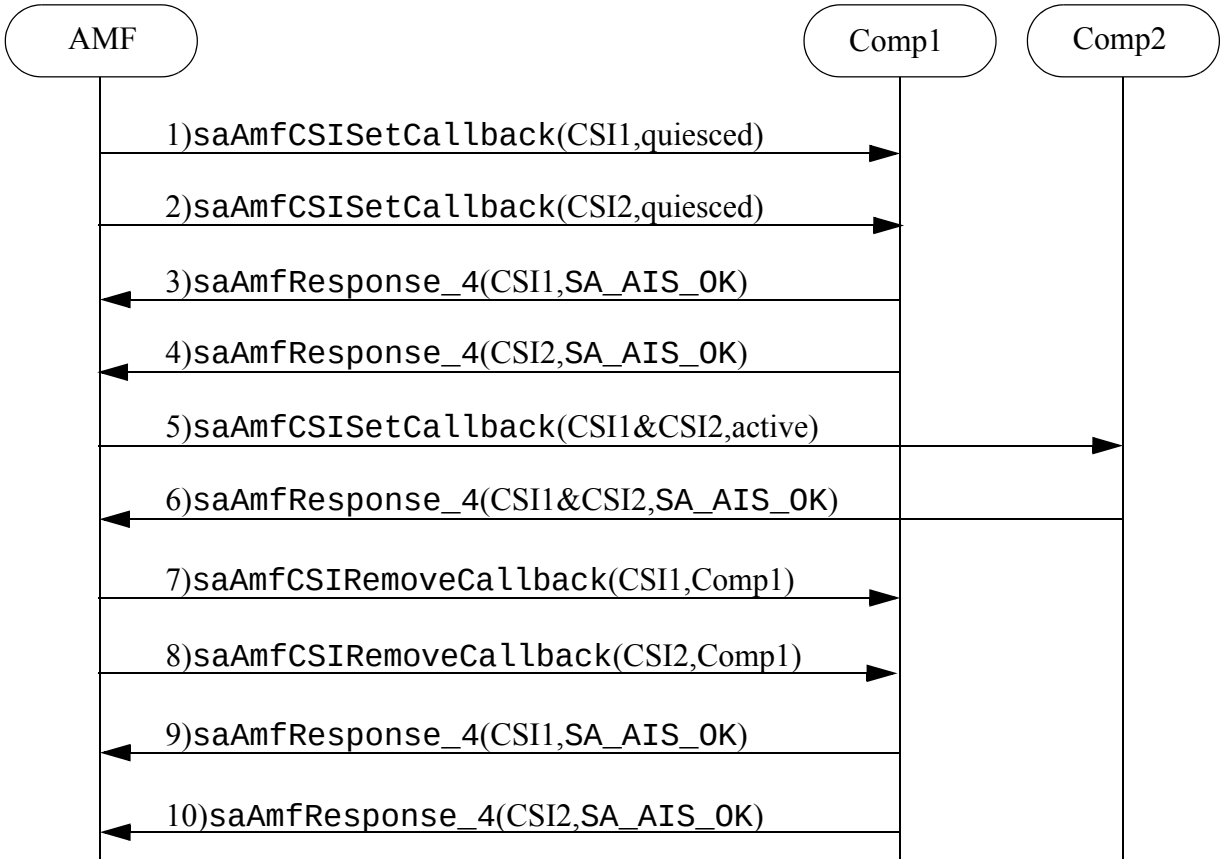


Notice that as only one of the SIs has been locked, only the component service instance corresponding to that SI is manipulated.

10.5 Administrative Lock of a Service Unit

The context of this scenario is a service group with 2N redundancy model having two service units with a single regular SA-aware, contained, or proxy component each. Two SIs are assigned to the two service units such that component Comp1 and component Comp2 have each two CSI assignments: Comp1 is assigned active for CSI1 and CSI2, and Comp2 is assigned standby for CSI1 and CSI2. The following diagram shows the sequence of actions when one of the service units (the one having components assigned active for CSI1 and CSI2) is administratively locked.

FIGURE 44 Administrative Lock of a Service Unit for the 2N Case



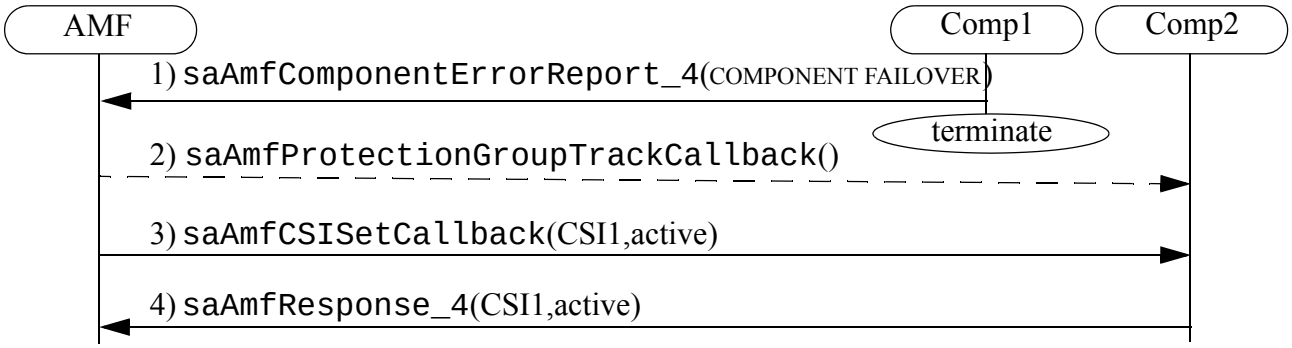
The Availability Management Framework should use the `csiFlags` value `SA_AMF_TARGET_ALL` in the callback of step 5 to guarantee that 2N semantics are honored.

Note that the same sequence shown in FIGURE 44 applies when a service unit is locked as a consequence of a node lock administrative action. In the example, it is assumed that the other service unit in the service group resides on another node.

10.6 A Simple Fail-Over

The context of this scenario is a service group with 2N redundancy model having two service units with a single regular SA-aware, contained, or proxy component each. A single SI is assigned such that component Comp1 and component Comp2 have each a single CSI assignment (CSI1): Comp1 is assigned active for CSI1, and Comp2 is assigned standby for CSI1. The following diagram shows Comp1 disabled by a fault, and the Availability Management Framework responding by assigning the active HA state to Comp2 for CSI1.

FIGURE 45 Fail-Over Scenario for a Service Group with the 2N Redundancy Model



The dotted line indicates an optional transaction. Note that the protection group call-back informs the registered component that Comp1 left the protection group.

10.7 Administrative Shutdown of an SI Having a Container CSI

This scenario shows the sequence of operations when a service instance having a container CSI is shut down. The container component and the associated contained components have different redundancy models.

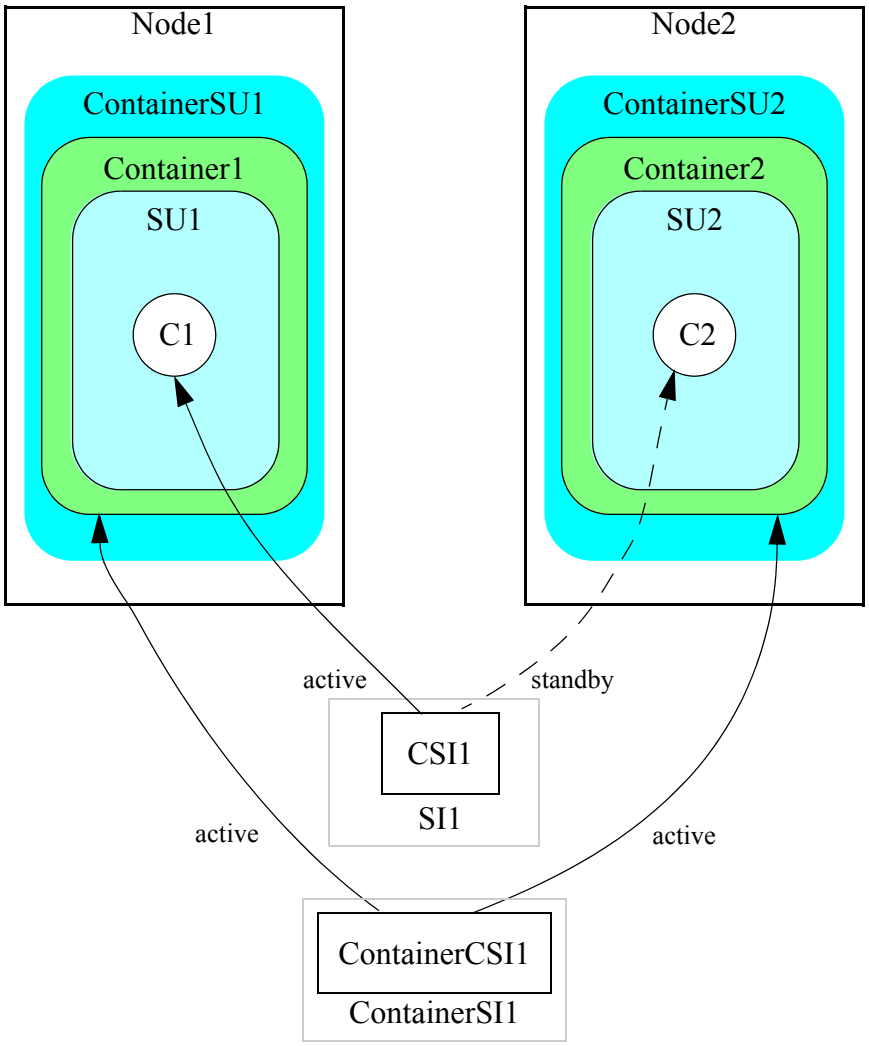
Service Group for Service Units Containing the Container Components

- It has an N-way active redundancy model and contains the service units ContainerSU1 and ContainerSU2, which are configured on Node1 and Node2, respectively.
- ContainerSU1 contains the component Container1, and ContainerSU2 contains Container2.
- The service instance ContainerSI1 is assigned to the service group. ContainerSI1 contains ContainerCSI1.

Service Group for Service Units Containing the Contained Components

- It has a 2N redundancy model and contains the service units SU1 and SU2, which are configured on Node1 and Node2, respectively.
- SU1 contains the component C1, and SU2 contains C2.
- The service instance SI1 is assigned to the service group. SI1 contains the component service instance CSI1.
- In the Availability Management Framework configuration, C1 and C2 are configured with `saAmfCompContainerCsi` set to ContainerCSI1.

FIGURE 46 Scenario for Shutting Down a Service Instance Having a Container CSI

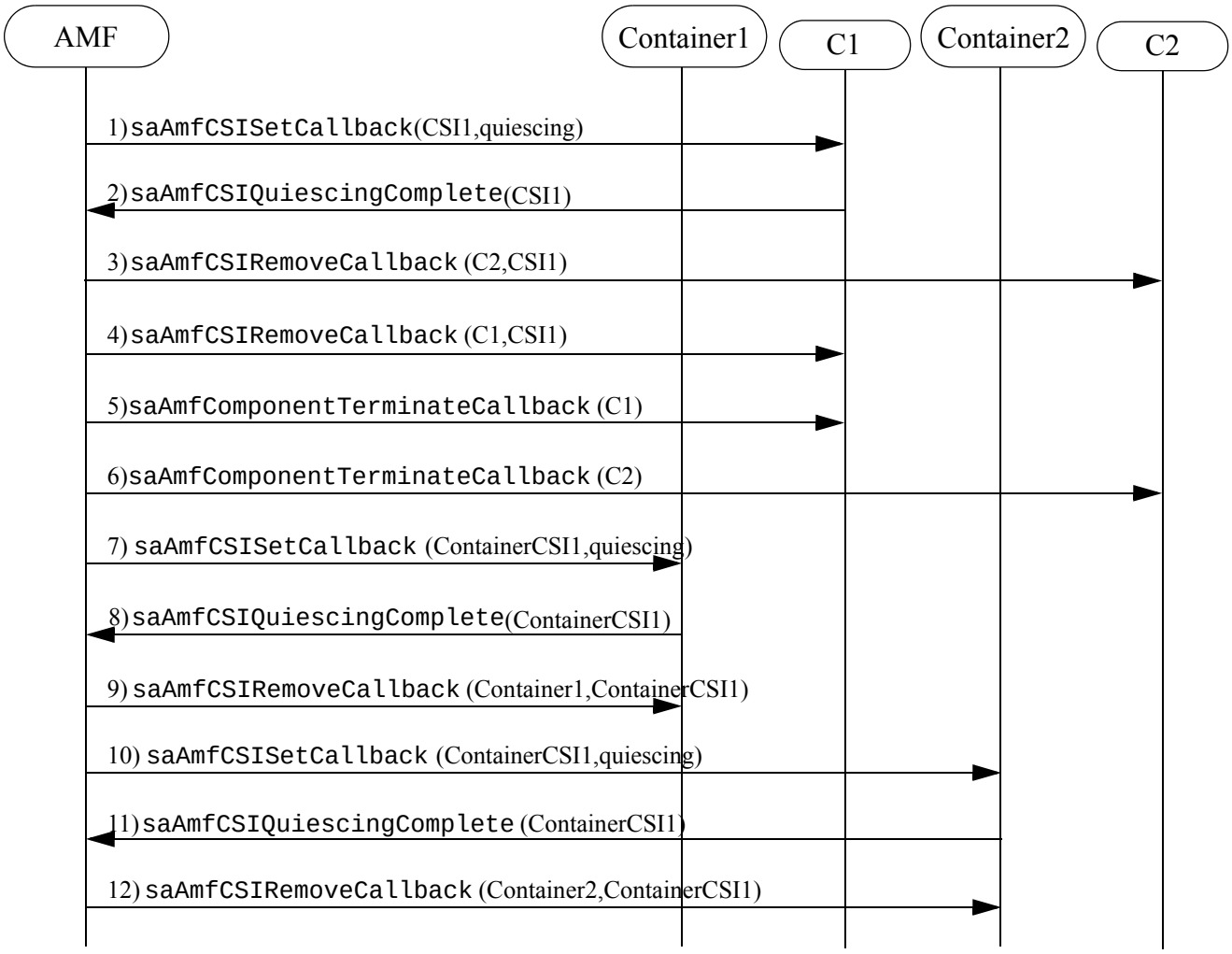


The following sequence diagram shows the sequence of operations when ContainerSI1 is shut down. For readability purposes, the invocations of `saAmfResponse_4()` to respond to Availability Management Framework requests are not shown in the diagram.

Before the shutdown administrative operation is issued, the state is as follows:

- Container1 and Container2 have the active HA state for ContainerCSI1.
- Container1 handles the life cycle of C1, and Container2 handles the life cycle of C2.
- For CSI1, C1 has the active HA state and C2 has the standby HA state.

FIGURE 47 Administrative Shutdown of a Service Instance Having a Container CSI



The main transitions are:

- C1 is set to quiescing for CSI1.
- After the quiescing is completed, CSI1 is removed from C1 and C2.
- C1 and C2 are terminated.
- Container1 is set to quiescing for ContainerCS1; after the quiescing is completed, ContainerCSI1 is removed from Container1.
- Container2 is set to quiescing for ContainerCS1; after the quiescing is completed, ContainerCSI1 is removed from Container2.

10.8 Administrative Lock of an SI Having a Container CSI

This scenario shows the sequence of operations when a service instance having a container CSI is locked. The container component and the associated contained components have different redundancy models.

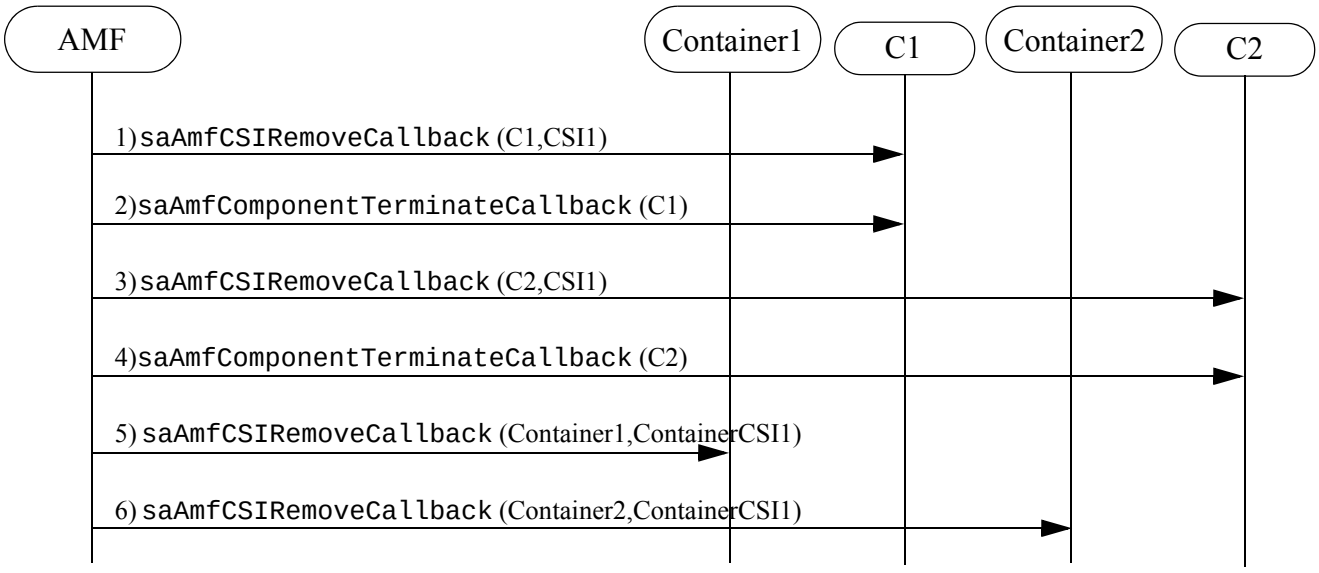
The configuration of the service group for the container components and of the service group for the contained components is the same as for Section 10.7. FIGURE 46 in Section 10.7 depicts this scenario.

The following sequence diagram shows the sequence of operations when ContainerSI1 is locked. For readability purposes, the invocations of `saAmfResponse_4()` to respond to Availability Management Framework requests are not shown in the diagram.

Before the lock administrative operation is issued, the state is as follows:

- Container1 and Container2 have the active HA state for ContainerCSI1.
- Container1 handles the life cycle of C1, and Container2 handles the life cycle of C2.
- For CSI1, C1 has the active HA state and C2 has the standby HA state.

FIGURE 48 Administrative Lock of a Service Instance Having a Container CSI



The main transitions are:

- CSI1 is removed from C1, and C1 is terminated.
- CSI1 is removed from C2, and C2 is terminated.
- ContainerCSI1 is removed from C1 and from C2.

10.9 Administrative Lock of a Service Unit with a Container Component

This scenario shows the sequence of operations when a service unit containing a container component is locked. The container component and the associated contained components have different redundancy models.

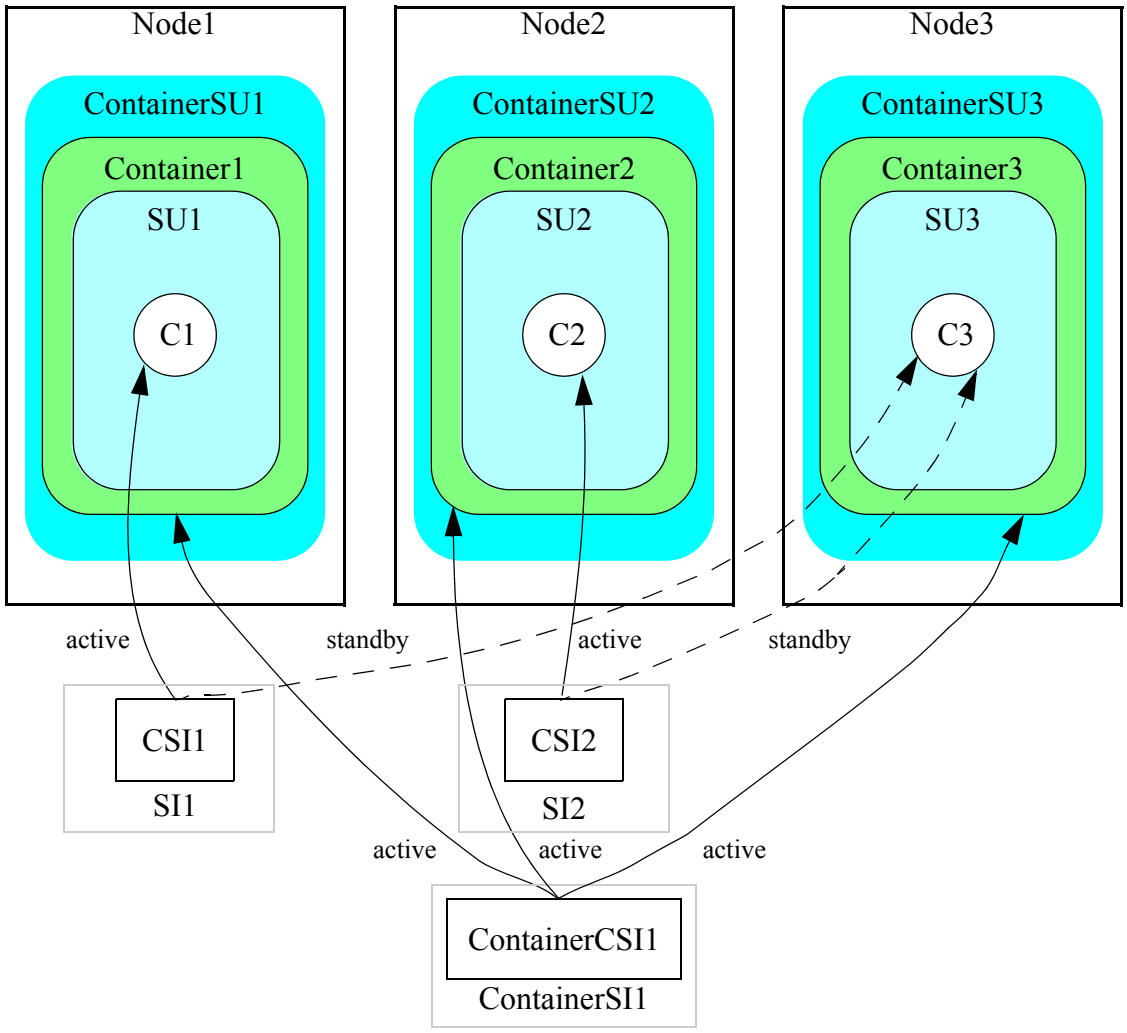
Service Group for Service Units Containing the Container Components

- It has a N-way active redundancy model and contains the service units ContainerSU1, ContainerSU2, and ContainerSU3, which are configured on Node1, Node2, and Node3, respectively.
- ContainerSU1 contains the component Container1, ContainerSU2 contains Container2, and ContainerSU3 contains Container3.
- The service instance ContainerSI1 is assigned to the service group. ContainerSI1 contains the component service instance ContainerCSI1.

Service Group for Service Units Containing the Contained Components

- It has a 2+1 (N+M) redundancy model and contains the service units SU1, SU2, and SU3, which are configured on Node1, Node2, and Node3, respectively.
- SU1 contains the component C1, SU2 contains C2, and SU3 contains C3.
- The service instances SI1 and SI2 are assigned to the service group. SI1 contains the component service instance CSI1, and SI2 contains the component service instance CSI2.
- C1, C2, and C3 are configured with `saAmfCompContainerCsi` set to ContainerCSI1.

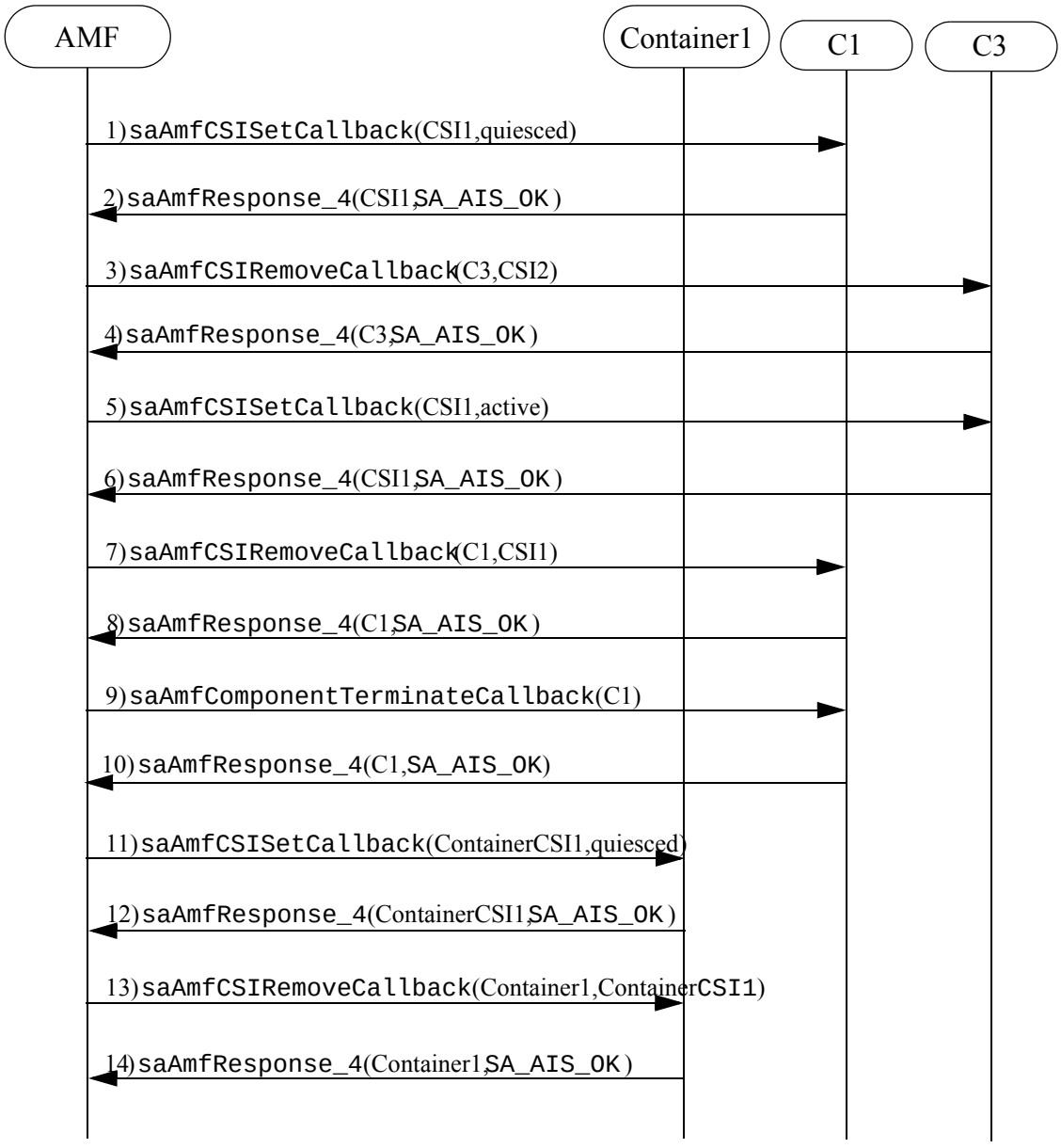
FIGURE 49 Scenario for Locking a Service Unit Containing a Container Component



The following sequence diagram shows the sequence of operations when ContainerSU1 is locked. Before the lock administrative operation is issued, the state is as follows:

- Container1, Container2, and Container3 have the active HA state for ContainerCSI1.
- Container1, Container2, and Container3 handle the life cycle of C1, C2, and C3, respectively.
- C1 has the active HA state for CSI1, C2 has the active HA state for CSI2, and C3 has the standby HA state for CSI1 and CSI2.

FIGURE 50 Administrative Lock of a Service Unit Containing a Container Component



The main transitions are:

- C1 is quiesced for CSI1.
- The standby HA state for CSI2 is removed from C3.
- C3 is set active for CSI1.
- CSI1 is removed from C1.

- C1 is terminated.
- Container1 is quiesced for ContainerCSI1.
- ContainerCSI1 is removed from Container1.

10.10 Restart of a Container Component

This scenario shows the restart of a container component Container1 that is handling the life cycle of a single contained component C1 as a result of the restart administrative operation on the container component.

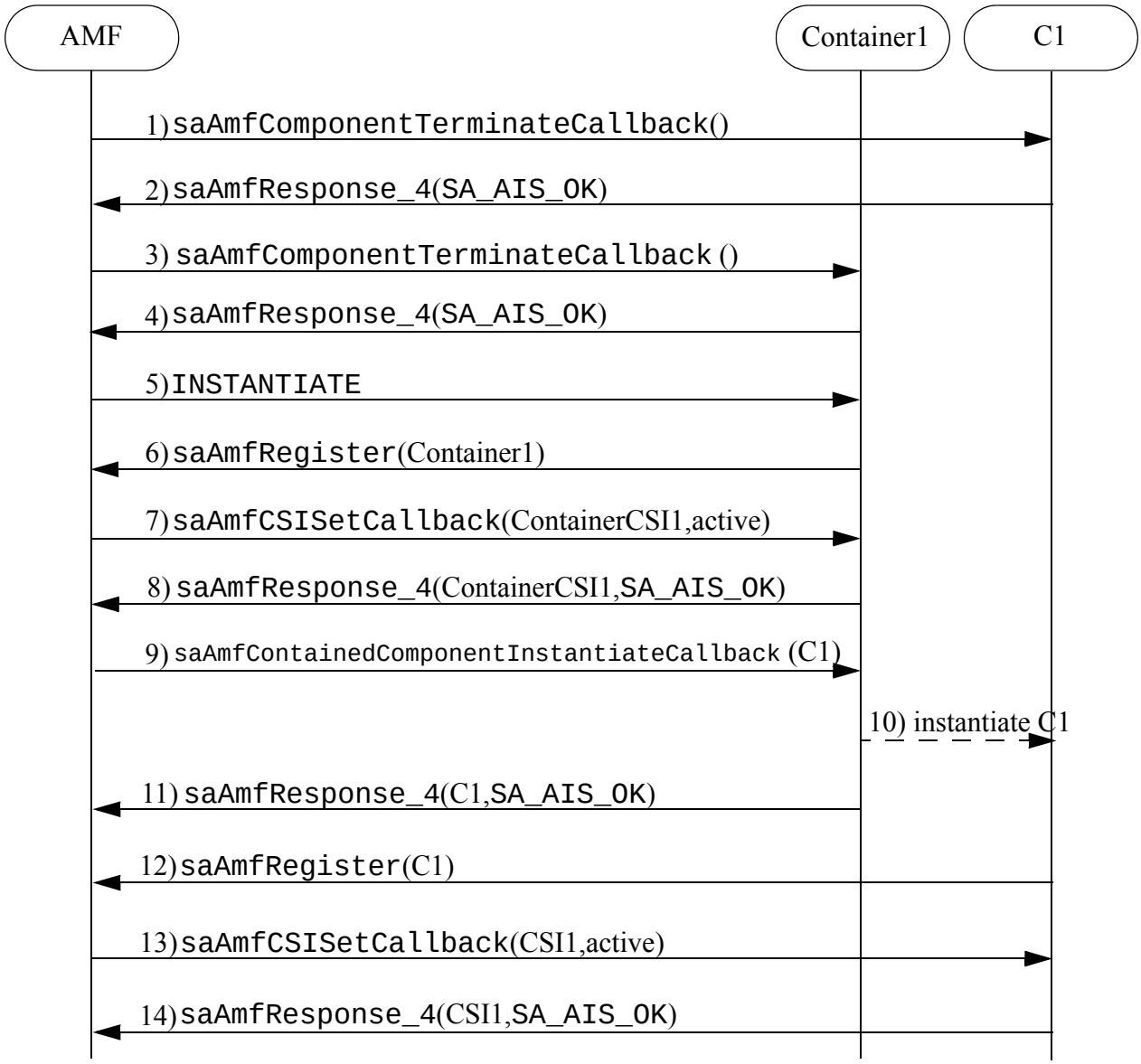
In the Availability Management Framework configuration, the ContainerCSI1 component service instance is configured to be assigned to Container1 such that Container1 can handle the life cycle of C1. No other CSIs are assigned to Container1.

The single CSI1 component service instance is configured to be assigned to C1.

This scenario applies regardless of the configuration of service units containing these components and regardless of the redundancy models used for the service groups containing contained components, provided that both the container component and the contained components have been configured with the `saAmfCompDisableRestart` configuration attribute set to SA_FALSE (see Section 3.2.2.1).

The restart of Container1 is described in the following sequence diagram. It is assumed that before the restart administrative operation is issued on Container1, Container1 is assigned active for ContainerCSI1, and C1 is assigned active for CSI1.

FIGURE 51 Restart of a Container Component



The main transitions are:

- The contained component C1 is terminated; then, Container1 is terminated.
- The Availability Management Framework runs the INSTANTIATE command to instantiate Container1.
- Container1 registers with the Availability Management Framework.

- The Availability Management Framework assigns Container1 active for ContainerCSI1.
- Container1 responds to the Availability Management Framework that it is ready to provide service for ContainerCSI1.
- The Availability Management Framework invokes the `saAmfContainedComponentInstantiateCallback()` callback function of Container1 to instantiate C1.
- Container1 instantiates C1 through a private interface.
- C1 registers with the Availability Management Framework.
- The Availability Management Framework assigns C1 active for CSI1.

11 Alarms and Notifications

The Availability Management Framework produces alarms and notifications to convey important information regarding the operational and functional state of the objects under its control to an administrator or a management system.

These reports vary in perceived severity and include alarms, which potentially require an operator intervention, and notifications which signify important state or object changes. A management entity should regard notifications, but they do not necessarily require an operator intervention.

The vehicle to be used for producing alarms and notifications is the Notification Service of the Service Availability™ Forum (abbreviated as NTF, see [3]), and hence the various notifications are partitioned into categories, as described in this service.

In some cases, this specification uses the word “Unspecified” for values of attributes that the vendor is at liberty to set to whatever makes sense in the vendor’s context, and the SA Forum has no specific recommendation regarding such values. Such values are generally optional from the CCITT Recommendation X.733 perspective (see [10]).

11.1 Setting Common Attributes

The following attributes of the notifications presented in Section 11.2 are not shown in their description, as the generic description presented here applies to all of them:

- Notification Id - Depending on the Notification Service function used to send the notification, this attribute is either implicitly set by the Notification Service or provided by the caller.
- Notifying Object - DN of the entity generating the notification. This name must conform to the SA Forum AIS naming convention and must contain at least the `safApp` RDN value portion of the DN set to the specified standard RDN value of the SA Forum AIS Service generating the notification, that is `safAmfService`. For details on the AIS naming convention, refer to [2].

The following notes apply to all Availability Management Framework notifications presented in Section 11.2:

- Correlated Notifications - Correlation Ids are supplied to correlate notifications that have been generated because of a related cause. The correlated notifications attribute should include

  ⇒ in the first position the root notification identifier of the related tree of notifications as described in the Notification Service specification (see [3]);

⇒ in the second position the parent notification identifier of the same tree;

⇒ in the third position the notification identifier of the sibling notification, if any. This sibling notification is the opening pair of the current notification such as the alarm that is being cleared or the start of an administrative operation or a configuration change that has ended.

If any of these notifications is unknown, the SA_NTF_IDENTIFIER_UNUSED value must be used. This value may be omitted in trailing positions.

- Event Time - If not specified in the description of a particular notification, this attribute contains the time when the Availability Management Framework detected the event leading to the notification.
- Notification Class Identifier - The `vendorId` portion of the `SaNtfClassIdT` data structure must be set to SA_NTF_VENDOR_ID_SAF always, and the `majorId` field must be set to SA_SVC_AMF (as defined in the `SaServicesT` enumeration in [2]) for all notifications that follow the standard formats described in this specification. The `minorId` field will vary based on the specific notification.

11.2 Availability Management Framework Notifications

The following subsections describe the notifications that an Availability Management Framework implementation shall produce.

11.2.1 Availability Management Framework Alarms

11.2.1.1 Component Instantiation Failed

Description

The Availability Management Framework was unable to successfully instantiate a particular component. This means that

- either the `INSTANTIATE` command executed on the component either returned an error exit status or failed to successfully complete within the time period specified by the configured timeout, or
- the corresponding callback invoked on the component or on its proxy or associated container component returned an error code other than `SA_AIS_OK` or failed to successfully complete within the configured timeout, and
- and all subsequent attempts by the Availability Management Framework to revive the component, including a possible node reboot, did not resolve the issue.

As a consequence, the component will enter the instantiation-failed presence state. For more details, refer to Section 4.6.

Clearing Method

Manual, after taking the appropriate administrative action.

Table 23 Component Instantiation Failed Alarm

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ALARM_PROCESSING`
Notification Object	Mandatory	LDAP DN of the component whose instantiation failed
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_COMP_INSTANTIATION_FAILED`, see Section 7.4.11.1 on page 261
Additional Text	Optional	“Instantiation of Component <LDAP DN of component> failed”
Additional Information	Mandatory	`infoId` = `SA_AMF_NODE_NAME`, `infoType` = `SA_NTF_VALUE_LDAP_NAME`, `infoValue` = LDAP DN of node on which the component is hosted
Probable Cause	Mandatory	Applicable value from enum `SaNtfProbableCauseT` in [3]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum `SaNtfSeverityT` in [3]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

11.2.1.2 Component Cleanup Failed

Description

The Availability Management Framework was unable to successfully cleanup a particular component after failing to successfully terminate the component. Under such circumstances, the component enters the termination-failed presence state. This condition could potentially cause a service disruption, as the workload (assigned to the failed component) would not be reassigned to some other healthy component because of redundancy model constraints, requiring an administrator to take a corrective action in order to recover. For more details, refer to Section 4.8.

Clearing Method

Manual, after taking the appropriate administrative action.

Table 24 Component Cleanup Failed Alarm

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ALARM_PROCESSING`
Notification Object	Mandatory	LDAP DN of the component whose cleanup failed
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_COMP_CLEANUP_FAILED`, see Section 7.4.11.1 on page 261
Additional Text	Optional	“Cleanup of Component <LDAP DN of component> failed”
Additional Information	Mandatory	`infoId` = `SA_AMF_NODE_NAME`, `infoType` = `SA_NTF_VALUE_LDAP_NAME`, `infoValue` = LDAP DN of node on which the component is hosted
Probable Cause	Mandatory	Applicable value from enum `SaNtfProbableCauseT` in [3]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum `SaNtfSeverityT` in [3]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

11.2.1.3 Cluster Reset Triggered by a Component Failure

Description

A component failed and recommended to the Availability Management Framework the SA_AMF_CLUSTER_RESET cluster reset recovery action.

Clearing Method

(1) Manual, after taking the appropriate administrative action or

(2) issue an implementation-specific optional alarm with perceived severity SA_NTF_SEVERITY_CLEARED to convey that the cluster reset was successful.

Table 25 Cluster Reset Triggered by a Component Failure Alarm

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ALARM_PROCESSING`
Notification Object	Mandatory	LDAP DN of the component that recommended an `SA_AMF_CLUSTER_RESET` recovery
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_CLUSTER_RESET`, see Section 7.4.11.1 on page 261
Additional Text	Optional	“Failure of Component <LDAP DN of component> triggered cluster reset.”
Additional Information	Optional	Unspecified
Probable Cause	Mandatory	Applicable value from enum `SaNtfProbableCauseT` in [3]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum `SaNtfSeverityT` in [3]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

11.2.1.4 Service Instance Unassigned

Description

A particular unit of work indicated by a service instance has no active assignments to any service unit, which is potentially causing a service disruption. In other words, the service instance transitioned to the unassigned assignment state, as explained in Section 3.2.3.2.

This alarm is typically generated when the Availability Management Framework is unable to successfully execute a recovery to prevent the service disruption and maintain service availability in case of a failure (node, service unit, and so on). This alarm should be also generated when an administrative action renders a service instance unassigned.

Clearing Method

Manual, after taking the appropriate administrative action.

Table 26 Service Instance Unassigned Alarm

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ALARM_PROCESSING`
Notification Object	Mandatory	LDAP DN of the service instance that has no current active assignments
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_SI_UNASSIGNED`, see Section 7.4.11.1 on page 261
Additional Text	Optional	“SI designated by <LDAP DN of the SI> has no current active assignments to any SU.”
Additional Information	Optional	Unspecified
Probable Cause	Mandatory	Applicable value from enum `SaNtfProbableCauseT` in [3]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum `SaNtfSeverityT` in [3]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

11.2.1.5 Proxy Status of a Component Changed to Unproxied

Description

This alarm is generated by the Availability Management Framework when it reliably confirms that a component that was previously being proxied has currently no proxy component mediating for it, that is, the Availability Management Framework has not been able to engage another component to assume the mediation responsibility for a component whose proxy component has failed. The proxied component has now the `SA_AMF_PROXY_STATUS_UNPROXIED` status, as defined in Section 7.4.4.8.

See also Section 11.2.2.7.

Clearing Method

Manual, after taking the appropriate administrative action.

Table 27 Proxy Status of a Component Changed to Unproxied Alarm

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ALARM_PROCESSING`
Notification Object	Mandatory	LDAP DN of component that is no longer proxied.
Notification Class Identifier	NTF -Internal	`minorId` = `SA_AMF_NTFID_COMP_UNPROXIED`, see Section 7.4.11.1 on page 261
Additional Text	Optional	Unspecified
Additional Information	Optional	Unspecified
Probable Cause	Mandatory	Applicable value from enum `SaNtfProbableCauseT` in [3]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum `SaNtfSeverityT` in [3]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

11.2.2 Availability Management Framework State Change Notifications

11.2.2.1 Administrative State Change Notify

Description

The administrative state of a node, a service unit, a service group, a service instance, an application, or the cluster changed.

Table 28 Administrative State Change Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the logical entity whose administrative state changed
Notification Class Identifier	NTF-Internal	The following values for the `minorid` from Section 7.4.11.1 on page 261 apply: node: `SA_AMF_NTFID_NODE_ADMIN_STATE` SU: `SA_AMF_NTFID_SU_ADMIN_STATE` SG: `SA_AMF_NTFID_SG_ADMIN_STATE` SI: `SA_AMF_NTFID_SI_ADMIN_STATE` application: `SA_AMF_NTFID_APP_ADMIN_STATE` cluster: `SA_AMF_NTFID_CLUSTER_ADMIN_STATE`
Additional Text	Optional	Unspecified
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	`SA_NTF_MANAGEMENT_OPERATION`
Changed State Attribute ID	Optional	`SA_AMF_ADMIN_STATE`
Old Attribute Value	Optional	Applicable value from enum `SaAMFAdminStateT`
New Attribute Value	Mandatory	Applicable value from enum `SaAMFAdminStateT`

11.2.2.2 Operational State Change Notify

Description

The operational state of a node or a service unit changed.

Table 29 Operational State Change Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the logical entity whose operational state changed
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_NODE_OP_STATE` for node and `minorid` = `SA_AMF_NTFID_SU_OP_STATE` for SU, see Section 7.4.11.1 on page 261
Additional Text	Optional	Unspecified

Additional Information	Optional	`infoId` = `SA_AMF_MAINTENANCE_CAMPAIGN_DN`
		`infoType` = `SA_NTF_VALUE_LDAP_NAME`
		`infoValue` = LDAP DN of the upgrade campaign, that is, the contents of the `saAmfSUMaintenanceCampaign` attribute

Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_AMF_OP_STATE`
Old Attribute Value	Optional	Applicable value from enum `SaAmfOperationalStateT`
New Attribute Value	Mandatory	Applicable value from enum `SaAmfOperationalStateT`

11.2.2.3 Presence State Change Notify

Description

The presence state change of a service unit is reported only if it becomes instantiated, uninstantiated, or restarting.

Table 30 Presence State Change Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the service unit whose presence state changed
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_SU_PRESENCE_STATE`, see Section 7.4.11.1 on page 261
Additional Text	Optional	Unspecified
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_AMF_PRESENCE_STATE`
Old Attribute Value	Optional	Unspecified
New Attribute Value	Mandatory	Applicable value from enum `SaAmfPresenceStateT`

11.2.2.4 HA State Change Notify

Description

The HA state of a service unit for an assigned service instance changes.

Table 31 HA State Change Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the service unit whose HA state for a particular SI changed
Notification Class Identifier	NTF- Internal	`minorId` = `SA_AMF_NTFID_SU_SI_HA_STATE`, see Section 7.4.11.1 on page 261
Additional Text	Optional	“The HA state of SI <LDAP DN> assigned to SU <LDAP DN> changed.”
Additional Information	Mandatory	`infoId` = `SA_AMF_SI_NAME`, `infoType` = `SA_NTF_VALUE_LDAP_NAME`, `infoValue` = LDAP DN of the SI for which the HA state of the SU changed
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_AMF_HA_STATE`
Old Attribute Value	Optional	Unspecified
New Attribute Value	Mandatory	Applicable value from enum `SaAmfHAStateT`

11.2.2.5 HA Readiness State Change Notify

Description

The HA readiness state of a service unit for a service instance changes.

Table 32 HA Readiness State Change Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the service unit whose HA readiness state for a particular SI changed
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_SU_SI_HA_READINESS_STATE`, see Section 7.4.11.1 on page 261
Additional Text	Optional	“The HA readiness state of SI <LDAP DN> assigned to SU <LDAP DN> changed.”
Additional Information	Mandatory	`infoId` = `SA_AMF_SI_NAME`, `infoType` = `SA_NTF_VALUE_LDAP_NAME`, `infoValue` = LDAP DN of the SI for which the HA readiness state of the SU has changed
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_AMF_HA_READINESS_STATE`
Old Attribute Value	Optional	Unspecified
New Attribute Value	Mandatory	Applicable value from enum `SaAmfHAReadinessStateT`
Number of Correlated Notifications	Mandatory	2
Correlated Notifications	Mandatory	Root and parent correlation IDs passed to `saAmfHAReadinessStateSet()`

11.2.2.6 SI Assignment State Change Notify

Description

The assignment state of a service instance changed. This notification is generated for all assignment state transitions for a service instance, except when the assignment state changes to SA_AMF_ASSIGNMENT_UNASSIGNED in which case an alarm is generated, as explained in Section 11.2.1.4 on page 425.

Table 33 SI Assignment State Change Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	SA_NTF_OBJECT_STATE_CHANGE
Notification Object	Mandatory	LDAP DN of the service instance whose assignment state changed
Notification Class Identifier	NTF-Internal	minorId = SA_AMF_NTFID_SI_ASSIGNMENT_STATE, see Section 7.4.11.1 on page 261
Additional Text	Optional	“The Assignment state of SI <LDAP DN of SI> changed.”
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	SA_NTF_OBJECT_OPERATION or SA_NTF_UNKNOWN_OPERATION
Changed State Attribute ID	Optional	SA_AMF_ASSIGNMENT_STATE
Old Attribute Value	Optional	Applicable value from enum SaAmfAssignmentStateT
New Attribute Value	Mandatory	Applicable value from enum SaAmfAssignmentStateT

11.2.2.7 Proxy Status of a Component Changed to Proxied

Description

This notification is generated by the Availability Management Framework when it could engage another component to assume the mediation responsibility for a proxied component which was in the SA_AMF_PROXY_STATUS_UNPROXIED status (see Section 7.4.4.8). The proxied component assumes then the SA_AMF_PROXY_STATUS_PROXIED status.

See also Section 11.2.1.5.

Table 34 Proxy Status of a Component Changed to Proxied Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	SA_NTF_OBJECT_STATE_CHANGE
Notification Object	Mandatory	LDAP DN of the proxied component whose proxy failed and is currently not being proxied
Notification Class Identifier	NTF-Internal	minorId = SA_AMF_NTFID_COMP_PROXY_STATUS, see Section 7.4.11.1 on page 261
Additional Text	Optional	Unspecified
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	SA_NTF_OBJECT_OPERATION or SA_NTF_UNKNOWN_OPERATION
Changed State Attribute ID	Optional	SA_AMF_PROXY_STATUS
Old Attribute Value	Optional	SA_AMF_PROXY_STATUS_UNPROXIED
New Attribute Value	Mandatory	SA_AMF_PROXY_STATUS_PROXIED

11.2.3 Availability Management Framework Notifications of Miscellaneous Type

11.2.3.1 Error Report Notification

Description

This notification is generated by the Availability Management Framework when the the `saAmfComponentErrorReport_4()` function is invoked.

Table 35 Error Report Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ERROR_REPORT`
Notification Object	Mandatory	LDAP DN of the component on which the error has been reported
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_ERROR_REPORT`, see Section 7.4.11.1 on page 261
Event Time	Mandatory	Value of the `errorDetectionTime` parameter of the `saAmfComponentErrorReport_4()` function
Number of Correlated Notifications	Mandatory	0, 1, 2, or more, depending on the values of the different fields passed to the `saAmfComponentErrorReport_4()` function and on the number of error report notifications generated for the same error condition (see also Section 11.1)
Correlated Notifications	Mandatory	`rootCorrelationId` and `parentCorrelationId` passed to the `saAmfComponentErrorReport_4()` function, as applicable, and the identifier of any error report notification generated for the same error condition (see also Section 11.1)
`numAdditionalInfo` field of `SaNtfNotificationHeaderT` (see [3])	Mandatory	1
Additional Information	Mandatory	`infoId` = `SA_AMF_AI_RECOMMENDED_RECOVERY`, `infoType` = `SA_NTF_VALUE_UINT64`, `infoValue` = `recommendedRecovery` parameter of the `saAmfComponentErrorReport_4()` function}
Additional Information	Optional	`infoId` = `SA_AMF_AI_APPLIED_RECOVERY`, `infoType` = `SA_NTF_VALUE_UINT64`, `infoValue` = recovery engaged by the Availability Management Framework (`SaAmfRecommendedRecoveryT`)

11.2.3.2 Error Clear Notification

Description

This notification is generated by the Availability Management Framework when the `saAmfComponentErrorClear_4()` function is invoked.

Table 36 Error Clear Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_ERROR_CLEAR`
Notification Object	Mandatory	LDAP DN of the component that has been repaired
Notification Class Identifier	NTF-Internal	`minorId` = `SA_AMF_NTFID_ERROR_CLEAR`, see Section 7.4.11.1 on page 261
Event Time	Mandatory	Time at which the `saAmfComponentErrorClear_4()` function is invoked
Number of Correlated Notifications	Mandatory	0, 1, 2, or more, depending on the values of the different fields passed to the `saAmfComponentErrorClear_4()` function and on the number of error report notifications generated for the same error condition (see also Section 11.1)
Correlated Notifications	Mandatory	`rootCorrelationId` and `parentCorrelationId` passed to the `saAmfComponentErrorClear_4()` function, as applicable, and the identifier of any error report notification generated for the same error condition (see also Section 11.1)
`numAdditionalInfo` field of `SaNtfNotificationHeaderT` (see [3])	Mandatory	0

Appendix A Implementation of CLC Interfaces

The commands or callbacks used to control the life cycle of the various component categories differ considerably. To describe conveniently these life cycle operations, the specification uses the names instantiate, terminate, and cleanup for these operations. The following table shows how these operations are implemented:

Table 37 Implementation of CLC Operations for Each Component Category

Component Category	Operation	Implementation
SA-aware (excluding contained component)	`instantiate`	CLC-CLI `INSTANTIATE`
	`terminate`	`saAmfComponentTerminateCallback()`
	`cleanup`	CLC-CLI `CLEANUP`
contained component	`instantiate`	`saAmfContainedComponentInstantiateCallback()`
	`terminate`	`saAmfComponentTerminateCallback()`
	`cleanup`	`saAmfContainedComponentCleanupCallback()`
proxied, pre-instantiable	`instantiate`	`saAmfProxiedComponentInstantiateCallback()`
	`terminate`	`saAmfComponentTerminateCallback()`
	`cleanup`	CLC-CLI `CLEANUP` (except for external) `saAmfProxiedComponentCleanupCallback()`
proxied, non-pre-instantiable	`instantiate`	`saAmfCSISetCallback()`
	`terminate`	`saAmfCSIRemoveCallback()`
	`cleanup`	CLC-CLI `CLEANUP` (except for external) `saAmfProxiedComponentCleanupCallback()`
non-proxied, non-SA-aware	`instantiate`	CLC-CLI `INSTANTIATE`
	`terminate`	CLC-CLI `TERMINATE`
	`cleanup`	CLC-CLI `CLEANUP`

If both an `saAmfProxiedComponentCleanupCallback()` callback and a `CLEANUP` command are defined for local components, the callback is invoked. The `CLEANUP` command is run only if the callback returns an error; however, in situations where the Availability Management Framework needs to abruptly terminate all components on a node that left the cluster membership, the `CLEANUP` command is run directly without invoking the callback.

Appendix B API Functions and Registered Processes

A process that has not registered any component may only invoke a subset of the Availability Management Framework API functions, and only a subset of the Availability Management Framework callback functions may be invoked for the process. For each API function of the Availability Management Framework (sorted alphabetically) and for all the remaining SA Forum API functions in the last line as a whole, the following table indicates by a 'YES' in the second column whether the function can be invoked in the context of any process. A 'NO' in the second column indicates that the function can be invoked only in the context of a registered process for a component.

Table 38 API Functions and Registered Processes

API Interfaces	API Can be Invoked in the Context of Any Process
`saAmfComponentErrorClear_4()`	YES
`saAmfComponentErrorReport_4()`	YES
`saAmfComponentNameGet()`	YES
`saAmfComponentRegister()`[1]	NO
`SaAmfComponentTerminateCallbackT`	NO
`SaAmfContainedComponentCleanupCallbackT`	NO
`SaAmfContainedComponentInstantiateCallbackT`	NO
`saAmfCorrelationIdsGet()`	NO
`saAmfCSIQuiescingComplete()`	NO
`SaAmfCSIRemoveCallbackT`	NO
`SaAmfCSISetCallbackT`	NO
`saAmfDispatch()`	YES
`saAmfFinalize()`	YES
`SaAmfHAReadinessStateSet()`	NO
`saAmfHAStateGet()`	YES
`SaAmfHealthcheckCallbackT`	YES
`saAmfHealthcheckConfirm()`	YES

Table 38 API Functions and Registered Processes (Continued)

API Interfaces	API Can be Invoked in the Context of Any Process
`saAmfHealthcheckStart()`	YES
`saAmfHealthcheckStop()`	YES
`saAmfInitialize_4()`	YES
`saAmfPmStart_3()`	YES
`saAmfPmStop()`	YES
`SaAmfProtectionGroupNotificationFree_4()`	YES
`saAmfProtectionGroupTrack_4()`	YES
`SaAmfProtectionGroupTrackCallbackT_4`	YES
`saAmfProtectionGroupTrackStop()`	YES
`SaAmfProxiedComponentCleanupCallbackT`	NO
`SaAmfProxiedComponentInstantiateCallbackT`	NO
`saAmfResponse_4()`	YES
`saAmfSelectionObjectGet()`	YES
All SA Forum API functions and callbacks	YES

1. The process of an SA-aware component calling this function to register the component becomes the registered process for the component only after this function completes successfully.
The process of a proxy component calling this function to register a proxied component becomes the registered process for the proxied component only after this function completes successfully.

Appendix C Example for Proxy/Proxied Association

The following example outlines the procedure by which a proxied component gets associated with a proxy component as well as the subsequent interactions of these components with the Availability Management Framework during the instantiation and registration phase. The example uses two service groups with different redundancy models, one containing the proxied components (SGx1) and the other one containing the proxy components (SGx2).

Proxied SGx1:

- has a 2+1 (N+M) redundancy model,
- and contains the service units SUx1, SUx2, and SUx3.
- SUx1 contains component cx1, SUx2 contains component cx2, and SUx3 contains component Cx3.
- CSIs corresponding to the components cx1, cx2, and cx3 are CSIx1, CSIx2, and CSIx3, respectively.

Proxy SGp1:

- has a 2N redundancy model,
- and contains the service units SUp1 and SUp2.
- SUp1 contains component cp1 and SUp2 contains component cp2.
- The CSI corresponding to the components is CSIp1 (“Proxy CSI”)
- There is only a single SI, SIx1, which is protected by this service group.

The Availability Management Framework configuration will have the following CSI associations for the proxied components in SGx1:

- cx1 should be proxied by CSIp1
- cx2 should be proxied by CSIp1
- cx3 should be proxied by CSIp1

When the Availability Management Framework instantiates SGp1, it may decide by some logic that CSIp1 should be assigned active to cp1 and standby to cp2. The decision is based on the configuration data and HA requirements; the fact that CSIp1 is a proxy CSI is not taken into account during its decision; however, when CSIp1 is assigned to cp1 as active, the Availability Management Framework has the following information at that time:

- CSIp1 is associated with the proxied components cx1, cx2, and cx3. This information is derived from the configuration.

- CSIp1 is currently being assigned active to cp1.

Hence, the Availability Management Framework concludes that cp1 is currently supposed to “proxy” proxied components cx1, cx2, and cx3, and it starts instantiating them.

The following steps illustrate an instantiation sequence for this sample configuration when cx1 and cx2 are instantiated and registered but cx3 does not register (potentially because of a failure).

1. AMF runs the INSTANTIATE command to instantiate cp1.
2. cp1 registers with AMF by invoking saAmfComponentRegister().
3. AMF assigns cp1 active for CSIp1 by invoking SaAmfCSISetCallbackT.
4. AMF invokes SaAmfProxiedComponentInstantiateCallbackT for cx1 on cp1.
5. cp1 registers cx1 with AMF by invoking saAmfComponentRegister().
6. cp1 returns SA_AIS_OK to AMF for step 4. by invoking saAmfResponse_4().
7. AMF invokes SaAmfProxiedComponentInstantiateCallbackT for cx2 on cp1.
8. cp1 registers cx2 with AMF by invoking saAmfComponentRegister().
9. cp1 returns SA_AIS_OK to AMF for step 7. by invoking saAmfResponse_4().
10. AMF invokes SaAmfProxiedComponentInstantiateCallbackT for cx3 on cp1.
11. cx3 is not registered with AMF by cp1 for some reason (for instance, failure).
12. cp1 returns failure for step 10. (see step 15. for subsequent AMF actions).
13. AMF assigns CSIx1 to cx1 by invoking SaAmfCSISetCallbackT of cp1.
14. AMF assigns CSIx2 to cx2 by invoking SaAmfCSISetCallbackT of cp1.
15. AMF invokes SaAmfProxiedComponentCleanupCallbackT for cx3 on cp1 and carries out the regular procedure to try to revive cx3. If it fails, AMF transitions cx3 to the instantiation-failed presence state, raises the alarm, and so on.

Note: In the scenario described above, CSIx3 was never assigned, which would also have been the case for an SA-aware component.

Appendix D Interaction with CLM

The Availability Management Framework must use the track APIs of the Cluster Membership Service (CLM) to be notified about changes in the cluster membership (see [4]). Depending on the cause of the membership change, the Availability Management Framework may be notified at different steps during the change:

- `SA_CLM_CHANGE_VALIDATE`: when invoked during this step, the Availability Management Framework has the possibility to reject the pending change. The Availability Management Framework must reject the pending change if it would bring the assignment state of some service instances to unassigned.
- `SA_CLM_CHANGE_START`: when invoked during this step, the Availability Management Framework must perform any action that prepares for the pending change.
  If the cause of the pending change is a shutdown administrative operation performed on an entity that hosts an AMF node (PLM hardware element, execution environment, or CLM node), the Availability Management Framework must perform the same operations it would perform in case of an AMF node shutdown administrative operation (see Section 9.4.6); the Availability Management Framework then terminates all service units hosted by this node and only replies to the cluster membership track callback when all these actions are completed.

  If the cause of the pending change is a lock administrative operation performed on an entity that hosts an AMF node (PLM hardware element, execution environment, or CLM node) or an eviction of a PLM hardware element that hosts an AMF node, the Availability Management Framework must perform the same operations it would perform in case of an AMF node lock administrative operation (see Section 9.4.3); the Availability Management Framework then terminates all service units hosted by this node and only replies to the cluster membership track callback when all these actions are completed.
- `SA_CLM_CHANGE_COMPLETED`: when invoked during this step, the Availability Management Framework must perform any action that is still necessary to react to the change that just occurred. No action may be needed in this step if all required actions have already been performed during a previous `SA_CLM_CHANGE_START` step.
  If a node has left the membership, the Availability Management Framework must terminate all service units hosted by this AMF node, which implies the termination and cleanup of all processes belonging to components. The Availability Management Framework must also check all service groups affected by the termination of these service units and must perform any adjustment to match their preferred configuration. These adjustments may involve reassigning service instances to other service units and instantiating some uninstantiated service

units.
If the notification indicates a pending failure of a node, the Availability Management Framework must attempt to reassign all service instances currently assigned to service units hosted on that node to service units hosted on other nodes.
If a node has joined the membership, the Availability Management Framework must check whether any service units must be instantiated on the node and whether some service instances must be assigned to them.

Index of Definitions

Numerics
1_active component capability 107
1_active_or_1_standby component capability 107
1_active_or_y_standby component capability 107
2N redundancy model
  see also redundancy models
  definition 122
  auto-adjust option 123
  ordered list of service units for a service group 122
  preferred number of in-service service units 122

A
abrupt termination of a component 72
active assignment of a component 79
active assignment of a service unit 68
active assignment of/for a component service instance 79
active assignment of/for a service instance 68
active HA state of a component for a component service instance 78
active HA state of a service unit for a service instance 67
active-active redundancy configuration 161
administrative state of a cluster 93
administrative state of a node 90
administrative state of a service group 89
administrative state of a service instance 87
administrative state of an application 92
AM_START command 215
AM_STOP command 215
AMF cluster 39
AMF node 38
AMF node capacity 115
application type 56
applications
  definition 56
  administrative state
    definition 92
    locked 92
    locked-instantiation 92
    shutting-down 92
    unlocked 92
  type 56
assigned service units 111
assignment state of a service instance 88
associated contained component 45
associated container component 45
auto-adjust option 112, 123, 137, 152, 163, 177
auto-adjust probation period 114
automatic repair 197
Availability Management Framework cluster 39
Availability Management Framework node 38

C
capacity 115
CLC-CLI
  definition 207
  arguments 210
  commands
    AM_START 215
    AM_STOP 215
    CLEANUP 214
    INSTANTIATE 211
    TERMINATE 213
  environment variables 209
  exit status 210
  pathname of a command 208
  pathname prefix 208
  per-command pathname 208
CLC-CLI arguments 210
CLC-CLI environment variables 209
CLEANUP command 214
CLM cluster 39
CLM node 38
cluster
  administrative state
    definition 93
    locked 93
    locked-instantiation 93
    shutting-down 93
    unlocked 93
  reset 40
  start 40
Cluster Membership cluster 39
Cluster Membership node 38
cluster reset 40
cluster start 40
collocated contained components 45
component capability
  model 107
  1_active 107
  1_active_or_y_standby 107
  1_active_or_1_standby 107
  non-pre-instantiable 107
  x_active 107
  x_active_and_y_standby 107
  x_active_or_y_standby 107
component capability model 107
component category 42
component healthchecks *see* healthchecks
component life cycle *see* CLC-CLI
component monitoring
  definition 190
  external active monitoring 190
  internal active monitoring 190
  passive monitoring 190
component or service unit fail-over recovery 195
component service instance
  definition 50
  container CSI 51, 224
  fail-over 96
  name/value pairs 51
  Proxy CSI 51
  switch-over 96
component service instance fail-over 96
component service instance switch-over 96
component service type 52
component type 50

component-invoked healthchecks 233
components
  see also healthchecks
  definition 41
  abrupt termination 72
  active assignment 79
  active assignment of/for a component service instance 79
  associated contained 45
  associated container 45
  category 42
  collocated contained 45
  contained 44
  container 44
  external 42
  HA readiness state for a component service instance
    definition 84
    not-ready-for-active 85
    not-ready-for-assignment 86
    ready-for-active-degraded 85
    ready-for-assignment 85
  HA state for a component service instance
    definition 77
    active 78
    quiesced 78
    quiescing 78
    standby 78
  local 42
  non-pre-instantiable 49
  non-SA-aware 46
  operational state
    definition 75
    disabled 75
    enabled 75
  pre-instantiable 49
  presence state
    definition 71
    instantiated 72, 73
    instantiating 72
    instantiation-failed 72
    restarting 73
    terminating 72
    termination-failed 72
    uninstantiated 72
  proxied 46
  proxy 46, 48
  readiness state
    definition 76
    in-service 76
    out-of-service 76
    stopping 77
  regular SA-aware 44
  SA-aware 43
  standby assignment 79
  type 50
  workload 51
composite administrative operations 365
composite operations *see* composite administrative operations
contained components 44
container components 44
container CSI 51, 224
CSI *see* component service instance

D
dependencies
  dependencies amongst components 187
  dependency amongst component service instances 186
  SI-SI dependencies 185
  tolerance time of an SI-SI dependency 186
dependencies amongst components 187
dependency amongst component service instances 186
disabled operational state of a component 75
disabled operational state of a node 91
disabled operational state of a service unit 64

E
enabled operational state of a component 75
enabled operational state of a node 91
enabled operational state of a service unit 63
error detection 191
escalation of level 3 205
escalations of levels 1 and 2 203
exit status 210
external active monitoring 190
external components 42
external resources 42
external service units 53

F
fail-over
  component service instance fail-over 96
  node fail-over recovery 196, 199
  service instance fail-over 96
  service unit fail-over recovery 199
fail-over recovery 194
failover *see* fail-over
framework-invoked healthchecks 233
fully-assigned service instance 88

H
HA readiness state of a component for a component service instance 84
HA readiness state of a service unit for a service instance 69
HA state of a component for a component service instance 77
HA state of a service unit for a service instance 67
healthcheck key 232
healthcheck maximum-duration 234, 235
healthcheck period 234, 235, 236
healthchecks
  see also components
  definition 232
  component-invoked 233
  framework-invoked 233
  key 232
  maximum duration 234, 235
  period 234, 235, 236
  variants 233

I
in-service readiness state of a component 76
in-service readiness state of a service unit 65
in-service service unit 65
in-service service units 110
instantiable service units 110
INSTANTIATE command 211
instantiated presence state of a component 72, 73
instantiated presence state of a service unit 62
instantiated service units 111
instantiated spare service units 111
instantiating presence state of a component 72
instantiating presence state of a service unit 62
instantiation level 187
instantiation-failed presence state of a component 72
instantiation-failed presence state of a service unit 62
internal active monitoring 190

L
local components 42
local resources 41
local service units 53
locked administrative state of a cluster 93
locked administrative state of a node 90
locked administrative state of a service group 89
locked administrative state of a service instance 87
locked administrative state of a service unit 63
locked administrative state of an application 92
locked-instantiation administrative state of a cluster 93
locked-instantiation administrative state of a node 90
locked-instantiation administrative state of a service group 89
locked-instantiation administrative state of a service unit 63
locked-instantiation administrative state of an application 92
logical entities 37

M
maximum number of active SIs per service unit 136, 152, 163
maximum number of standby SIs per service unit 136, 152
monitoring *see* component monitoring
multiple (ranked) standby assignments 112

N
N+M redundancy model
  see also redundancy models
  definition 132
  auto-adjust option 137
  maximum number of active SIs per service unit 136
  maximum number of standby SIs per service unit 136
  ordered list of service units for a service group 135
  ordered list of SIs 135
  preferred number of active service units 136
  preferred number of in-service service units 136
  preferred number of standby service units 136
name/value pairs 51
no spare HA state 112
node capacity 115
node failfast recovery 196, 199
node fail-over recovery 196, 199
node group configuration attribute 57
node switch-over recovery 196, 199
nodes
  administrative state
    definition 90
    locked 90
    locked-instantiation 90
    shutting-down 90
    unlocked 90
  AMF 38
  capacity 115
  CLM 38
  failfast recovery 196
  fail-over recovery 199
  node group configuration attribute 57
  operational state
    definition 91
    disabled 91
    enabled 91
  switch-over recovery 196, 199
non-instantiated spare service units 111
non-pre-instantiable component capability 107
non-pre-instantiable components 49
non-pre-instantiable service unit 65
non-pre-instantiable service units 53
non-SA-aware 46
no-redundancy redundancy model
  see also redundancy models
  definition 175
  auto-adjust option 177
  ordered list of service units for a service group 177
  ordered list of SIs 177
  preferred number of in-service service units 177
not-ready-for-active HA readiness state of a component for a component service instance 85
not-ready-for-active HA readiness state of a service unit for a service instance 70
not-ready-for-assignment HA readiness state of a component for a component service instance 86
not-ready-for-assignment HA readiness state of a service unit for a service instance 71
N-way active redundancy model
  see also redundancy models
  definition 160
  active-active redundancy configuration 161
  auto-adjust option 163
  maximum number of active SIs per service unit 163
  ordered list of service units for a service group 162
  ordered list of SIs 162
  preferred number of active assignments per SI 162
  preferred number of assigned service units 162
  preferred number of in-service service units 162
  ranked service unit list per SI 162
N-way redundancy model
  see also redundancy models
  definition 149
  auto-adjust option 152
  maximum number of active SIs per service unit 152
  maximum number of standby SIs per service unit 152

ordered list of service units for a service group 151
ordered list of SIs 151
preferred number of assigned service units 152
preferred number of in-service service units 151
preferred number of standby assignments per SI 151
ranked service unit list per SI 151

O
operational state of a component 75
operational state of a node 91
operational state of a service unit 63
ordered list of service units for a service group 111, 122, 135, 151, 162, 177
ordered list of SIs 113, 135, 151, 162, 177
out-of-service readiness state of a component 76
out-of-service readiness state of a service unit 65
out-of-service service unit 65

P
partially-assigned service instance 88
passive monitoring 190
pathname of a CLC-CLI command 208
pathname prefix 208
per-command pathname 208
preferred number of active assignments per SI 162
preferred number of active service units 136
preferred number of assigned service units 152, 162
preferred number of in-service service units 122, 136, 151, 162, 177
preferred number of standby assignments per SI 151
preferred number of standby service units 136
pre-instantiable components 49
pre-instantiable service unit 65, 66
pre-instantiable service units 53
presence state of a component 71
presence state of a service unit 61
primitive administrative operations 365
primitive operations *see* primitive administrative operations
process
registered process for the component 230
protection group 56
proxied components 46
proxy component failure handling 219
proxy components 46, 48
proxy CSI 51

Q
quiesced HA state of a component for a component service instance 78
quiesced HA state of a service unit for a service instance 67
quiescing HA state of a component for a component service instance 78
quiescing HA state of a service unit for a service instance 67

R
rank 110
ranked list 110
ranked service unit list per SI 151, 162
ranking 110
readiness state of a component 76
readiness state of a service unit 64
ready-for-active-degraded HA readiness state of a component for a component service instance 85
ready-for-active-degraded HA readiness state of a service unit for a service instance 69
ready-for-assignment HA readiness state of a component for a component service instance 85
ready-for-assignment HA readiness state of a service unit for a service instance 69
recommended recovery actions 201
recovery 199
definition 192
escalation 201
fail-over
definition 194
component or service unit 195
node failfast 196, 199
node fail-over 196, 199
node switch-over 196, 199
service unit 199
restart
definition 193
restart all components of the service unit 193
restart the associated container component and collocated contained components 194
restart the erroneous component 193
recovery escalation 201
reduction procedure 112
redundancy level of a service instance 113
redundancy models
see also service groups, 2N redundancy model, N+M redundancy model, N-way redundancy model, N-way active redundancy model, no-redundancy redundancy model
definition 55
common definitions
auto-adjust option 112
auto-adjust probation period 114
in-service service units 110
instantiable service units 110
instantiated service units 111
instantiated spare service units 111
multiple (ranked) standby assignments 112
no spare HA state 112
non-instantiated spare service units 111
ordered list of service units for a service group 111
ordered list of SIs 113
reduction procedure 112
redundancy level of a service instance 113
node group configuration attribute 57
registered process for the component 230
regular SA-aware component 44
repair
definition 197
automatic repair 197
node failfast recovery 199
node fail-over recovery 199
node switch-over 199
service unit failover recovery 199

resources
  external 42
  local 41
restart 191
restart all components of the service unit 193
restart recovery 193
restart the associated container component and collocated contained components 194
restart the erroneous component 193
restarting presence state of a component 73
restarting presence state of a service unit 62
RM *see* redundancy models

S
SA-aware component 43
service
  component service instance 50
  type 55
service group redundancy model 109
service group type 55
service groups
  see also redundancy models
  definition 55
  administrative state
    definition 89
    locked 89
    shutting-down 89
    unlocked 89
  type 55
service instance 54
  administrative state
    definition 87
    fully-assigned 88
    locked 87
    partially-assigned 88
    shutting-down 87
    unassigned 88
    unlocked 87
  assignment state
    definition 88
  fail-over 96
  switch-over 96
  weight 115
service instance fail-over 96
service instance switch-over 96
service type 55
service unit failover recovery 199
service unit type 53
service units
  definition 52
  active assignment 68
  active assignment of/for a service instance 68
  administrative state
    locked 63
    locked-instantiation 63
    shutting-down 63
    unlocked 63
  assigned 111
  external 53
  HA readiness state for a service instance
    definition 69
    not-ready-for-active 70
    not-ready-for-assignment 71
    ready-for-active-degraded 69
    ready-for-assignment 69
  HA state for a service instance
    definition 67
    active 67
    quiesced 67
    quiescing 67
    standby 67
  in-service 110
  instantiable 110
  instantiated 111
  instantiated spare 111
  local 53
  non-instantiated spare 111
  non-pre-instantiable 53, 65
  operational state
    definition 63
    disabled 64
    enabled 63
  ordered list for a service group 111
  pre-instantiable 53, 65
  presence state
    definition 61
    instantiated 62
    instantiating 62
    instantiation-failed 62
    restarting 62
    terminating 62
    termination-failed 62
    uninstantiated 62
  readiness state
    definition 64
    in-service 65
    out-of-service 65
    stopping 66
  standby assignment 68
  type 53
SG *see* service groups
shutting-down administrative state of a cluster 93
shutting-down administrative state of a node 90
shutting-down administrative state of a service group 89
shutting-down administrative state of a service instance 87
shutting-down administrative state of a service unit 63
shutting-down administrative state of an application 92
SI *see* service instance
SI weight 115
SI-SI dependencies 185
spare
  instantiated spare service units 111
  non-instantiated spare service units 111
standby assignment 68, 79
standby HA state of a component for a component service instance 78
standby HA state of a service unit for a service instance 67
startup 40
stopping readiness state of a component 77

stopping readiness state of a service unit 66
SU *see* service unit
switch-over
  component service instance switch-over 96
  service instance switch-over 96

T
TERMINATE command 213
terminating presence state of a component 72
terminating presence state of a service unit 62
termination-failed presence state of a component 72
termination-failed presence state of a service unit 62
tolerance time of an SI-SI dependency 186

U
unassigned service instance 88
uninstantiated presence state of a component 72
uninstantiated presence state of a service unit 62
unlocked administrative state of a cluster 93
unlocked administrative state of a node 90
unlocked administrative state of a service group 89
unlocked administrative state of a service instance 87
unlocked administrative state of a service unit 63
unlocked administrative state of an application 92

V
variants of healthcheck 233

W
weights 115
workload 51
wrapper method 189

X
x_active component capability 107
x_active_and_y_standby component capability 107
x_active_or_y_standby component capability 107