Service Availability™ Forum Application Interface Specification

Availability Management Framework SAI-AIS-AMF-B.02.01

The Service Availability™ solution is high availability and more; it is the delivery of ultra-dependable communication services on demand and without interruption.

This Service Availability™ Forum Application Interface Specification document might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Current characterized errata are available on request.

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

The Service Availability™ Specification(s) (the "Specification") found at the URL http://www.saforum.org (the "Site") is generally made available by the Service Availability Forum (the "Licensor") 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 Specification are set forth in this agreement (this "Agreement").

IMPORTANT – PLEASE READ THE TERMS AND CONDITIONS PROVIDED IN THIS AGREEMENT BEFORE DOWNLOADING OR COPYING THE SPECIFICATION. IF YOU AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, CLICK ON THE "ACCEPT" BUTTON. BY DOING SO, YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS STATED IN THIS AGREEMENT. IF YOU DO NOT WISH TO AGREE TO THESE TERMS AND CONDITIONS, YOU SHOULD PRESS THE "CANCEL"BUTTON AND THE DOWNLOAD PROCESS WILL NOT PROCEED.

1. LICENSE GRANT. Subject to the terms and conditions of this Agreement, Licensor hereby grants you a non-exclusive, worldwide, non-transferable, revocable, but only for breach of a material term of the license granted in this section 1, fully paid-up, and royalty free license to:

a. reproduce copies of the Specification to the extent necessary to study and understand the Specification and to use the Specification to create products that are intended to be compatible with the Specification;

b. distribute copies of the Specification to your fellow employees who are working on a project or product development for which this Specification is useful; and

c. distribute portions of the Specification as part of your own documentation for a product you have built, which is intended to comply with the Specification.

2. DISTRIBUTION. If you are distributing any portion of the Specification in accordance with Section 1(c), your documentation must clearly and conspicuously include the following statements:

a. Title to and ownership of the Specification (and any portion thereof) remain with Service Availability Forum ("SA Forum").

b. The Specification is provided "As Is." SA Forum makes no warranties, including any implied warranties, regarding the Specification (and any portion thereof) by Licensor.

c. SA Forum shall not be liable for any direct, consequential, special, or indirect damages (including, without limitation, lost profits) arising from or relating to the Specification (or any portion thereof).

d. The terms and conditions for use of the Specification are provided on the SA Forum website.

3. RESTRICTION. Except as expressly permitted under Section 1, you may not (a) modify, adapt, alter, translate, or create derivative works of the Specification, (b) combine the Specification (or any portion thereof) with another document, (c) sublicense, lease, rent, loan, distribute, or otherwise transfer the Specification to any third party, or (d) copy the Specification for any purpose.

4. NO OTHER LICENSE. Except as expressly set forth in this Agreement, no license or right is granted to you, by implication, estoppel, or otherwise, under any patents, copyrights, trade secrets, or other intellectual property by virtue of your entering into this Agreement, downloading the Specification, using the Specification, or building products complying with the Specification.

5. OWNERSHIP OF SPECIFICATION AND COPYRIGHTS. The Specification and all worldwide copyrights therein are the exclusive property of Licensor. You may not remove, obscure, or alter any copyright or other proprietary rights notices that are in or on the copy of the Specification you download. You must reproduce all such notices on all copies of the Specification you make. Licensor may make changes to the Specification, or to items referenced

therein, at any time without notice. Licensor is not obligated to support or update the Specification.

6. WARRANTY DISCLAIMER. THE SPECIFICATION IS PROVIDED "AS IS." LICENSOR DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT OF THIRD-PARTY RIGHTS, FITNESS FOR ANY PARTICULAR PURPOSE, OR TITLE. Without limiting the generality of the foregoing, nothing in this Agreement will be construed as giving rise to a warranty or representation by Licensor that implementation of the Specification will not infringe the intellectual property rights of others.

7. PATENTS. Members of the Service Availability Forum and other third parties [may] have patents relating to the Specification or a particular implementation of the Specification. You may need to obtain a license to some or all of these patents in order to implement the Specification. You are responsible for determining whether any such license is necessary for your implementation of the Specification and for obtaining such license, if necessary. [Licensor does not have the authority to grant any such license.] No such license is granted under this Agreement.

8. LIMITATION OF LIABILITY. To the maximum extent allowed under applicable law, **LICENSOR DISCLAIMS ALL LIABILITY AND DAMAGES, INCLUDING DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, AND INCIDENTAL DAMAGES, ARISING FROM OR RELATING TO THIS AGREEMENT, THE USE OF THE SPECIFICATION OR ANY PRODUCT MANUFACTURED IN ACCORDANCE WITH THE SPECIFICATION, WHETHER BASED ON CONTRACT, ESTOPPEL, TORT, NEGLIGENCE, STRICT LIABILITY, OR OTHER THEORY. NOTWITHSTANDING ANYTHING TO THE CONTRARY, LICENSOR'S TOTAL LIABILITY TO YOU ARISING FROM OR RELATING TO THIS AGREEMENT OR THE USE OF THE SPECIFICATION OR ANY PRODUCT MANUFACTURED IN ACCORDANCE WITH THE SPECIFICATION WILL NOT EXCEED ONE HUNDRED DOLLARS ($100). YOU UNDERSTAND AND AGREE THAT LICENSOR IS PROVIDING THE SPECIFICATION TO YOU AT NO CHARGE AND, ACCORDINGLY, THIS LIMITATION OF LICENSOR'S LIABILITY IS FAIR, REASONABLE, AND AN ESSENTIAL TERM OF THIS AGREEMENT.**

9. TERMINATION OF THIS AGREEMENT. Licensor may terminate this Agreement, effective immediately upon written notice to you, if you commit a material breach of this Agreement and do not cure the breach within ten (30) days after receiving written notice thereof from Licensor. Upon termination, you will immediately cease all use of the Specification and, at Licensor's option, destroy or return to Licensor all copies of the Specification and certify in writing that all copies of the Specification have been returned or destroyed. Parts of the Specification that are included in your product documentation pursuant to Section 1 prior to the termination date will be exempt from this return or destruction requirement.

10. ASSIGNMENT. You may not assign, delegate, or otherwise transfer any right or obligation under this Agreement to any third party without the prior written consent of Licensor. Any purported assignment, delegation, or transfer without such consent will be null and void.

11. GENERAL. This Agreement will be construed in accordance with, and governed in all respects by, the laws of the State of Delaware (without giving effect to principles of conflicts of law that would require the application of the laws of any other state). You acknowledge that the Specification comprises proprietary information of Licensor and that any actual or threatened breach of Section 1 or 3 will constitute immediate, irreparable harm to Licensor for which monetary damages would be an inadequate remedy, and that injunctive relief is an appropriate remedy for such breach. All waivers must be in writing and signed by an authorized representative of the party to be charged. Any waiver or failure to enforce any provision of this Agreement on one occasion will not be deemed a waiver of any other provision or of such provision on any other occasion. This Agreement may be amended only by binding written instrument signed by both parties. This Agreement sets forth the entire understanding of the parties relating to the subject matter hereof and thereof and supersede all prior and contemporaneous agreements, communications, and understandings between the parties relating to such subject matter.

Table of Contents Availability Management Framework

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 Interface Specification (HPI) and with the Service Availability™ Forum System Management Specification.

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.01.0

(2) SAI-AIS-AMF-B.01.01

This section presents the changes of the current release, SAI-AIS-AMF-B.02.01, with respect to the SAI-AIS-AMF-B.01.01 release. Editorial changes are not mentioned here.

1.3.1 New Topics

- The “application” logical entity has been introduced in Section 3.2.7 on page 36. Section 3.3.7 on page 61 on the application states is also new in this release.
- The operational state of a node (see Section 3.3.6.2 on page 59) and the administrative state of a cluster (see Section 3.3.8 on page 61) are new in this release.
- 3.10 on page 133 presents approaches how to integrate non-SA-aware software or hardware entities into the Availability Management Framework model.

- Chapter 5 on page 153 presents the management of proxy and proxied components.
- Section 6.3.4 on page 170 contains the following new type definitions: *SaAmfReadinessStateT, SaAmfPresenceStateT, SaAmfOperationalStateT, SaAmfAdminStateT, SaAmfAssignmentStateT, SaAmfProxyStatusT,* and *SaAmfStateT*.
- New types for *saAmfRedandancyModelT, saAmfCompCapabilityModelT & saAmfCompCategoryT* has been introduced in *page 182*.
- Section 6.3.11 on page 182 defines notification-related types.
- Section 6.10.4 on page 222 for the *saAmfProtectionGroupNotificationFree()* function, which is used to release memory allocated by the Availability Management Framework library in the *saAmfProtectionGroupTrack()* function.
- Chapter 7 on page 229 describes the administrative API.
- Alarms and notifications are described in Chapter 9 on page 263.
- Appendix B on page 281 contains a table showing functions that can be invoked by or on unregistered processes.
- The concept of Auto Repair was added to 3.12.1.4 on page 139.

1.3.2 Clarifications

- Section 3.2.2 on page 28 clarifies that a process can only pertain to a component.
- A note in Section 3.3.2 on page 45 provides a general remark on the states defined for a proxied component.
- A paragraph after Table 13 on page 130 clarifies some cases for the x_active or 1_active capability models.
- Section 3.12.1.4 on page 139 treats the repair procedures in detail.
- The description of the INSTANTIATE command (see Section 4.4 on page 146) explains the actions taken by the Availability Management Framework when the INSTANTIATE command fails and either node reboot is disabled or a single reboot did not solve the problem.
- The description of the CLEANUP command has been extended. For details, refer to Section 4.6 on page 148.
- Section 6.1.1 on page 158 describes when a component may unregister with the Availability Management Framework. Refer also to related clarifications for the *saAmfComponentUnregister()* function in Section 6.5.2 on page 192.
- Section 6.1.2.4 on page 161 makes clear that the Availability Management Framework reports an error on the component if it does not receive a health-

check confirmation from the component before the end of every healthcheck period.

- Refer to the clarifications on the enums SA_AMF_PROTECTION_GROUP_NO_CHANGE and SA_AMF_PROTECTION_GROUP_STATE_CHANGE of the *SaAmfProtectionGroupChangesT* type (see Section 6.3.6.2 on page 178).
- The description of the *saAmfHealthcheckStart()* function (see Section 6.7.1 on page 200) clarifies that it is not possible to have at a given time and on the same *amfHandle* two healthchecks started for the same component name and health-check key.
- The description of the *saAmfComponentErrorReport()* function (see Section 6.11.1 on page 223) clarifies how the Availability Management Framework reacts to the setting of the *recommendedRecovery* parameter.
- Added a clarification to various callbacks regarding what needs to be done by the Availability Management Framework if the component fails to respond to a call-back or does not respond within a pre-configured time interval.

1.3.3 Changes in Return Values of API Functions

Table 1 Changes in Return Values of API Functions

API Function	Return Value	Change Type
saAmfComponentErrorClear()	SA_AIS_OK	clarified
saAmfComponentErrorClear()	SA_AIS_ERR_NOT_EXIST	changed
saAmfComponentErrorReport()	SA_AIS_ERR_NOT_EXIST	changed
saAmfComponentErrorReport()	SA_AIS_OK	clarified
saAmfComponentErrorReport()	SA_AIS_ERR_ACCESS	new
saAmfComponentRegister()	SA_AIS_ERR_INVALID_PARAM	extended
saAmfComponentRegister()	SA_AIS_ERR_BAD_OPERATION	changed
saAmfHealthcheckStart()	SA_AIS_ERR_ACCESS	new
saAmfHealthcheckStart()	SA_AIS_ERR_EXIST	changed
saAmfPmStart()	SA_AIS_ERR_ACCESS	new
saAmfProtectionGroupTrack()	SA_AIS_ERR_INIT SA_AIS_ERR_NO_SPACE SA_AIS_ERR_INVALID_PARAM	clarified
SaAmfProtectionGroupTrackCallbackT	SA_AIS_ERR_BAD_HANDLE SA_AIS_ERR_INVALID_PARAM SA_AIS_ERR_BAD_FLAGS SA_AIS_ERR_NOT_EXIST	new

1.3.4 Other Changes

- The new notion of “application” implies changes in the following descriptions:
  - Figure 2 on page 25
  - Table 3 on page 44
  - Section 3.9.1 on page 130
  - Section 3.9.1.1 on page 131
- Section 3.2.2.3 on page 31 describes the changes in the handling of proxy and proxied components.
- The definition of the presence state of a service unit (see Section 3.3.1.1 on page 40) has changed.

- The following sections have changed due to the addition of the new “Locked-instantiation” value to the administrative states:
  - Section 3.3.1.2 on page 41 on the administrative state of a service unit.
  - Section 3.3.3 on page 56 on service instance states.
  - Section 3.3.5 on page 58 on service group states.
  - Section 3.3.6 on page 59 on node states.
  - Section 3.4 on page 64 on fail-over and switch-over.
  - Table 11 on page 65 and Figure 5 on page 66.
  - Table 12 on page 67 and Figure 6 on page 68.
- The definition of the operational state (see Section 3.3.1.3 on page 42) and readiness state (see Section 3.3.1.4 on page 42) of a service unit have changed.
- Section 3.3.2.1 on page 46 on the presence state of a component treats the restart case in detail.
- Section 3.3.2.2 on page 48 on the operational state of a component describes that during a restart because of a failure, a component remains enabled and in-service, but its component service instances must be removed.
- Section 3.3.3.2 on page 56 describes the assignment state of a service instance. This state replaces the previous operational state of a service instance. This change has effect on other places in the specification such as Table 10 on page 62, Sections 3.9.1.1 on page 131 and 3.9.1.2 on page 131.
- Section 3.3.9 on page 62 and Table 10 on page 62 reflect the changes in the states supported for the logical entities.
- The “failback” notion has been renamed to “auto-adjust”.
- The row for “1_active_or_1_standby” in Table 13 on page 130 has changed.
- The restart of a service unit is explained in Section 3.12.1.2 on page 135.
- Note that the example contained in the topic component or service unit fail-over has changed for the N-way redundancy model (see Section 3.12.1.3 on page 136 on recovery).
- The configuration parameters described in Section 3.12.2.3 on page 144 on escalations of level 3 have changed from valid for all nodes to a per-node basis.
- The *SaAmfRecommendedRecoveryT* type definition (refer to Section 6.3.7 on page 180) has been extended by the SA_AMF_APPLICATION_RESTART enum. Note also the changed description of the SA_AMF_NO_RECOMMENDATION enum. The SA_AMF_CLUSTER_RESET case is described in detail. These changes are also shown in Section 3.12.2.1 on page 141.

- Memory allocated by the Availability Management Framework library in the *saAmfProtectionGroupTrack()* function (see Section 6.10.1 on page 216) must now be released by invoking the *saAmfProtectionGroupNotificationFree()* function (see Section 6.10.4 on page 222).

1.4 References

The following documents contain information that is relevant to specification:

[1] Service Availability™ Forum, Application Interface Specification, Overview, SAI-Overview-B.02.01

[2] Service Availability™ Forum, Application Interface Specification, Notification Service, SAI-AIS-NTF-A.01.01

[3] Service Availability™ Forum, Application Interface Specification, Cluster Membership Service, SAI-AIS-CLM-B.02.01

[4] Service Availability™ Forum, Application Interface Specification, Information Model Management Service (IMM), SAI-AIS-IMM-A.01.01

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

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

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

[8] 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 website (**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 Forum's website (**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 Forum's website (**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 Forum's website (**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 provides service availability by coordinating redundant resources within a cluster to deliver a system with no single point of failure.

The Availability Management Framework provides a view of one logical system 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. The Availability Management Framework drives the HA state and monitors the health of a component by invoking callback functions of the component, defined in this API. It manages internally also the readiness state, without exposing it to components. It further allows a component to query the Availability Management Framework for information about the component's HA state, using functions of the Availability Management Framework defined in this API.

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 physical entities that form the basis of the model, and the relationships between them, are described in section 3.1.

Section 3.2 discusses the logical entities managed by Availability Management Framework. The states and state models applicable to the relevant logical entities are discussed in Sections 3.3 and 3.5. Section 3.4 discusses fail-over and switch-over of service instances. The component capability model is discussed in Section 3.6. Section 3.7 describes the redundancy models supported by the Availability Management Framework in detail, and Section 3.8 investigates the interactions between the component capability model and the redundancy models. The remainder of the chapter discusses dependencies (Section 3.9), approaches for integrating legacy software and hardware in the framework (Section 3.10), component monitoring (Section 3.11), as well as error detection, recovery, repair, and escalation policy (Section 3.12).

3.1 Physical Entities

Every physical entity managed by the Availability Management Framework is a **resource**. These physical entities are either hardware equipment or software abstractions implemented by programs running on that hardware. These software abstractions include but are not limited to software processes, operating system features or operating system abstractions such as IP addresses or file systems like NFS.

A **physical node** is a particular type of resource that can behave like a computer, run a single instance of an operating system and export the SA Forum Application Interface Specification APIs. Physical nodes are interconnected with each other by some form of communication medium.

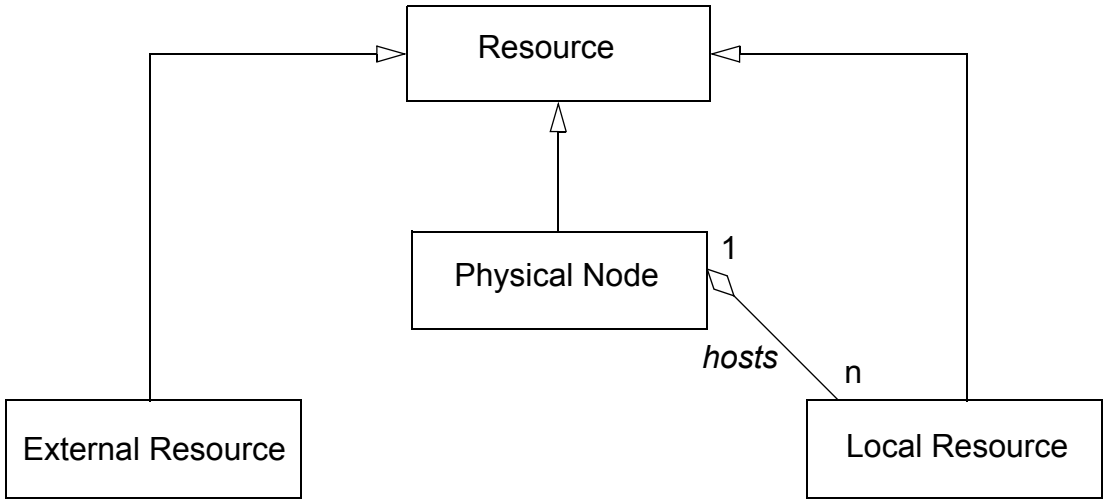
Resources that are contained, from a fault containment point of view, within a physical node are called **local resources**. This means that if a physical node fails, all of the local resources become inoperable. Local resources can be either software abstractions implemented by programs running on the physical node, or hardware equipment attached to the node (such as I/O devices), or the node itself.

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 processor board to operate can be modeled as an external resource. Failures of external resources are independent of physical node failures.

In the remainder of this specification usage of phrases like “resource external to the cluster” or “resources outside of the cluster” should be interpreted as “resource external to all nodes in the cluster”.

Figure 1 shows a UML diagram that depicts the physical entities of the system.

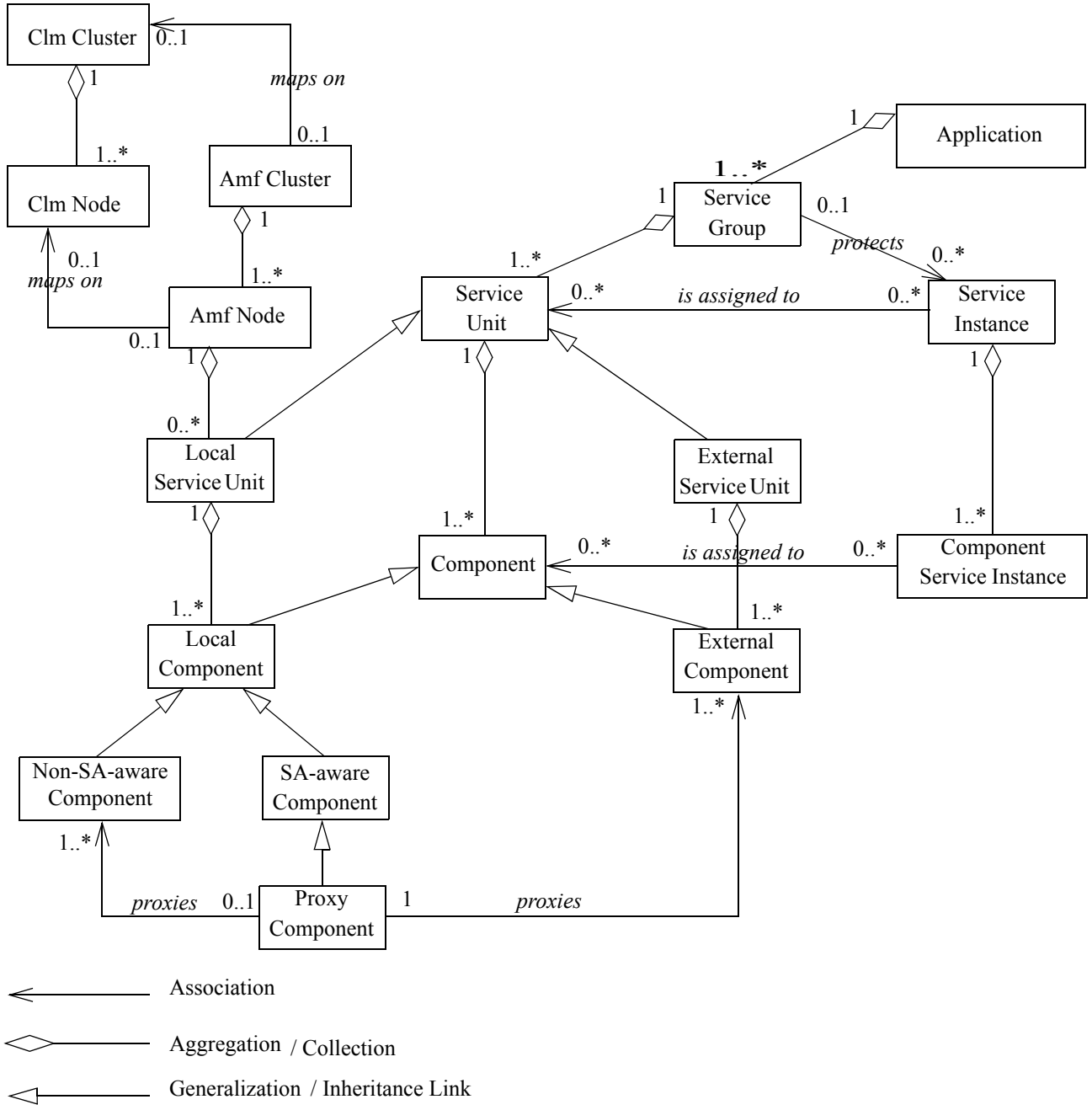
Figure 1 Physical Entities



3.2 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 the following Figure 2.

Figure 2 Logical entities



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; however, how this configuration is organized and how it is accessed and modified is not specified in this document. It is assumed that the Availability Management Framework obtains the cluster configuration from the configuration repository and is notified of any changes.

3.2.1 Cluster and Nodes

An Availability Management Framework node is the logical representation of a physical node that has been administratively configured in the Availability Management Framework configuration (Refer to the UML class description of SaAmfNode in [1] for a complete list of the attributes that is configured for an Availability Management Framework node). The configuration of an Availability Management Framework Node is valid even if

(a) there is no Cluster Membership node (the cluster node defined in [3]) mapped to the Availability Management Framework node or

(b) there is a Cluster Membership node mapped to the Availability Management Framework node, but the mapped Cluster Membership node is not in the cluster membership .

In both the cases above, the Availability Management Framework node cannot be used to provide service and none of the Availability Management Framework objects configured to be hosted by the Availability Management Framework node can be instantiated. An Availability Management Framework node is also a logical entity whose various states are managed by the Availability Management Framework. There are Availability Management Framework administrative operations that are defined for such nodes.

The complete set of Availability Management Framework nodes in the Availability Management Framework configuration defines the Availability Management Framework cluster. (See the UML class description of SaAmfCluster in [1] for details). Note that although the Availability Management Framework cluster and Cluster Membership cluster (defined in [3]) have a close relationship, they are not the same. Some Availability Management Framework nodes may not have associated Cluster Membership nodes (See SaClmNode class in UML model in [1]) and some Cluster Membership nodes may not have associated Availability Management Framework nodes. The procedure for mapping of an Availability Management Framework node to the corresponding Cluster Membership node is outside the scope of the Availability Management Framework specification. But the following general mapping rules apply:

(i) During cluster startup, it is possible that some Availability Management Framework nodes may be mapped to some Cluster Membership nodes by configuration, while other Availability Management Framework nodes are not mapped to configured Cluster Membership nodes.

(ii) Later during the life-span of the cluster, modifications may be made to the mapping of the Availability Management Framework node to the Cluster Membership node

The Availability Management Framework cluster is one of the entities that are under the Availability Management Framework control, and its various states are managed by the Availability Management Framework. There are Availability Management Framework administrative operations that are defined on the Availability Management Framework cluster.

However, the Availability Management Framework knows the association of its nodes to the Cluster Membership nodes and shall use this association to initiate operations such as rebooting a Cluster Membership node during a recovery operations.

If an Availability Management Framework node leaves the cluster membership, it is cleaned in the sense that no process using Availability Management Framework interfaces or no daemon that implements Availability Management Framework functionality is left over and all non-persistent Availability Management Framework information is deleted when the Availability Management Framework node rejoins the cluster membership. The Availability Management Framework can force a cluster node to leave the cluster membership by using a node reboot. It is required that the underlying cluster node of each Availability Management Framework node is equipped with an operating system providing a low level reboot interface. During a reboot, the cluster node leaves the cluster membership and rejoins after successful initialization.

In contrast, the restart of an Availability Management Framework node (see section 7.1.1 below) will only stop and start entities under Availability Management Framework control, without any impact on the cluster membership. The restart of the Availability Management Framework cluster (see Section 7.1.1) will restart all Availability Management Framework nodes and will also not impact the cluster membership. On the other hand, a **cluster reset** (see Section 6.3.7) reboots all Cluster Membership nodes of the cluster, where all nodes are first halted before any of the nodes boots again. In the remainder of this specification **cluster start** or **startup** is synonymous to the start of Availability Management Framework, which initially creates and instantiates the Availability Management Framework objects 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 Availability Management Framework nodes within the Availability Management Framework cluster. In order to make the rest of specification more readable and precise, we define the following notations:

- Throughout the specification, when the word "node" is used without an explicit qualification, it means "Availability Management Framework node".

- If "node" is used in the context of "reboot", "joining the cluster", and "leaving the cluster", it actually means the "the associated Cluster Membership node". For example, the sentence "the node will be rebooted" should be read as "the Cluster Membership node associate to the node will be rebooted". Similarly the sentence "When a node joins the cluster" should be interpreted as "When the Cluster Membership node associated with node joins the cluster."

Whenever "cluster" is used without an explicit qualification it assumes either of the following two meanings based on the context:

- Cluster as defined in membership service: for example, the sentence fragment "the node leaves the cluster", implies "the node leaves the cluster as defined in the membership service".
- A generic term that describes a set of nodes on which a set of highly available applications are deployed.

3.2.2 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 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 statements should be taken into account:

- The scope of a component must be small enough so that a failure of this 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 3.3.

The Availability Management Framework was primarily designed to manage local resources contained in nodes. This framework can also manage resources external to the cluster. 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 a single node.

- **External component**: An external component represents a set of resources that are external to the cluster.

Sections 3.2.2.1 to 3.2.2.4 describe how the Availability Management Framework manages local and external components. The information provided includes:

- An introduction to two sub-categories of components: Service Availability (SA)-aware and non-SA-aware components.
- The concepts of proxy and proxied components.
- A description of the component life cycle.

3.2.2.1 SA-Aware Components

High levels of service availability can only be attained if errors are detected, isolated, and recovered from, and failed entities repaired efficiently. Faster error recovery is possible if components have been chosen or are written so that they can register and interact with the Availability Management Framework to implement specific workload assignments and recovery policies. Such components must be designed so that the Availability Management Framework can dynamically assign 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 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. Processes of an SA-Aware component must exclusively belong only to that component. One of these processes registers the component with the Availability Management Framework through the *saAmfComponentRegister()* API function. This process, called the **registered process**, 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,
- assigning the HA state to the component for each workload.

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

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 descriptions of each API function given in Chapter 6 on page 157 explicitly mention when the function is restricted to a registered process. Additionally refer to Appendix B on page 281 for a table that displays which API/callback is restricted to registered processes only.

Note that legacy software running on a node that 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.

3.2.2.2 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, which 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 a proxy component mediates for are called **proxied components**.

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

However, 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 it must stop providing the service. Processes of a local non-SA-Aware component must exclusively belong to that component.

Application developers are encouraged to design applications, which 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 de-activation is usually performed through 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, there is minimal integration between the applications and the clustering middleware: 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 control of the Availability Management Framework, different functions can be modelled 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.

3.2.2.3 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 responsible for conveying requests made by the Availability Management Framework to its proxied components. The interactions between proxied components and their proxy component are private and not defined by this specification. The Availability Management Framework decides what proxied components a proxy component is responsible for 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 work load in the form of a component service instance (See Section 3.2.3 for the definition of component service instance).

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, another component (usually the component acting as standby of 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 there is no available proxy component to take over the mediation service, the Availability Management Framework loses control of these proxied components and becomes unaware of whether the proxied components are providing service.

Note that:

- A single proxy component can mediate between the Availability Management Framework and multiple proxied components.
- The redundancy model (refer to Section 3.7) 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.

3.2.2.4 Component Life Cycle

The Availability Management Framework directly controls the life cycle of non-proxied, local components through a set of command line interfaces 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 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 only start to provide a particular service 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 different component categories.

Table 2 Component Categories

Locality	HA Awareness	Proxy Property	Life Cycle Management
local	SA-aware	proxy or 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

3.2.3 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 3.3.2.4.

Each component service instance has a set of attributes (name/value pair), which characterize the workload assigned to the component. 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 **component service instance type.** All component service instances of the same type share the same list of attribute names. Several attributes with the same name may appear in the set of attributes of a component service instance, thus, providing support for multi-valued attributes.

3.2.4 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. Administrative operations are directed at service units as opposed to individual components.

A service unit can contain any number of components, but a given 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 3.3.

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 collocated on the same node. External service units can contain only external components. The external components represent resources that are external to the cluster.

Proxy components for local non-SA-aware components and these non-SA-aware components may reside in the same or in different service units. But a proxy component and its pre-instantiable proxied component cannot 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.

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**.

3.2.5 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 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 3.3.1.4), 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 it supports this particular type of 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 in a service instance may differ from the number of components aggregated in the service unit the service instance is assigned to. In such cases, some components may be left without any component service instance assignment while other components may have several component service instances assigned to them.

3.2.6 Service Groups

To ensure service availability in case of component failures, the Availability Management Framework manages redundant components that are contained in 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. 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 and be able to receive the assignment of any component service instance contained within the service instances protected by the service group.

The redundancy model defines how the service units in the service group are used to provide service availability. Refer to Section 3.7 on page 69 for details about service group redundancy models.

3.2.7 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.

Dependencies between service instances (described in Section 3.9.1 on page 130) are more common amongst service instances belonging to the application than amongst service instances of different applications.

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 contained components that 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.

3.2.8 Protection Groups

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

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

3.2.9 Service Unit Instantiation

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

- runs the INSTANTIATE command for all SA-aware components (including proxies),
- invokes the *saAmfProxiedComponentInstantiateCallback()* callback of the proxies of all pre-instantiable proxied components of the service unit,
- performs no action for non-pre-instantiable components. Such components are instantiated during the assignment of service instances to the service unit. (see Section 3.3.2.4 on page 50)

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

- invokes the *saAmfCSISetCallback()* callback of the proxies of all proxied components of the service unit,
- runs the INSTANTIATE command 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.

3.2.10 Illustration of Logical Entities

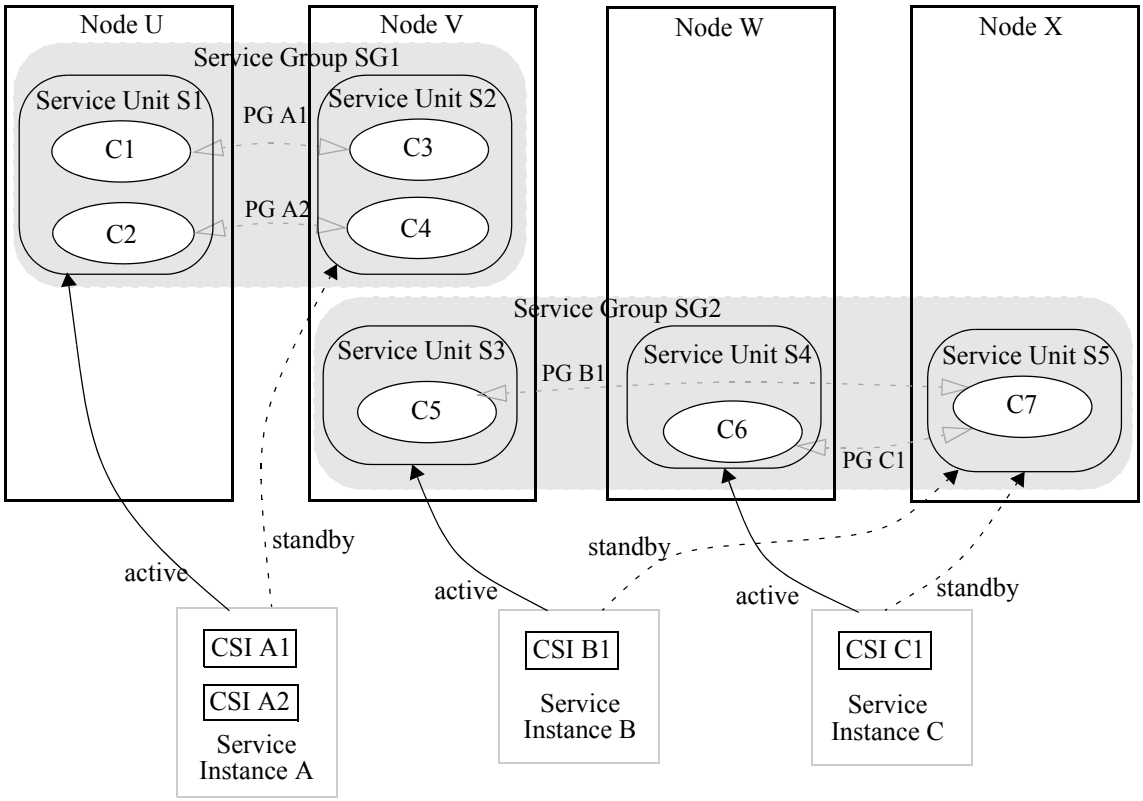
The example illustrated in Figure 3 below shows two service groups, SG1 and SG2. SG1 supports a single service instance (A) and SG2 supports two service instances (B and C).

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

Service units S1 and S2 each contain two components. Two component service instances (A1 and A2) are assigned to components C1 and C3, and to C2 and C4, respectively. 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 while protection group C1 contains components C6 and C7.

Figure 3 Elements of the System Model



3.3 State Models

The following sections describe the different states associated with service units, components, service instances, component service instances, service groups, and nodes. The Availability Management Framework API provides state management only for components and component service instances with 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 clear definition and extension of this specification.

3.3.1 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.

3.3.1.1 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 3.3.2.1.

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

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 3.3.1.5). 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 component's presence state is set to instantiation-failed, and the service unit's presence state is also set to instantiation-failed. If components were 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 to Section 4.5 and 4.6).

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 component's presence state is set to termination-failed and the service unit's presence state 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 only based 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 there are any errors in instantiating any of the constituent components of the service unit, the presence state becomes instantiation-failed. Similarly if there are errors in terminating any of the constituent components of the service unit the presence state becomes termination-failed.

3.3.1.2 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 ([3]). 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 3.3.5), the node (see Section 3.3.6.1), the application containing it (see Section 3.3.7), and the cluster (see Section 3.3.8) 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, and the service unit is currently 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 3.3.1.4) of that service unit.

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

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

3.3.1.3 Operational State

The **operational state** of the service unit refers to the ITU X.731 state management model ([5]). It is used by the Availability Management Framework to determine whether a service unit is capable of taking service instance assignments. The operational state of the service unit indicates whether the components within the service unit are operable or not. 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 3.12.1.4).
- **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.

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. It is again enabled after a successful repair. This is done 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 different from Availability Management Framework can perform the repair and declare the service unit repaired. When a restart recovery is executed in its scope, it is considered as an instantaneous combined recovery and repair action and hence its operational states remain enabled in such cases.

3.3.1.4 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**. This state indicates if a service unit is eligible to take service instance assignments from an administrative and health status viewpoint. This state is the only state 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 via the Availability Management Framework, but rather only indirectly, since 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 its operational state or the operational state of its containing node is disabled, or if its administrative state or the administrative state of its containing service group, node, application, or the cluster is either locked or locked-instantiation. The readiness state of a **pre-instantiable service unit** is out-of-service when 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. When the readiness state of a service unit is out-of-service, no new service instance can be assigned to it. If there are service instances assigned to the service unit at the time when it enters the out-of-service state, they are transferred to other service units (if possible) and removed.
- **in-service:** The readiness state of a **non-pre-instantiable service unit** is in-service if its operational state and the operational state of its containing node is enabled, and if its administrative state and the administrative states of its containing service group, node, application and the cluster are unlocked. 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 if its presence state is either instantiated or restarting. When a service unit is in the in-service readiness state, it is eligible for service instance assignments; however, it may not have been assigned any service instance yet.
- **stopping:** The service unit's readiness state is stopping if its operational state and the operational state of its containing node is enabled, if at least one of the administrative states of itself, the containing service group, node, application or the cluster are shutting-down, and none of these administrative states are locked or locked-instantiation. 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 3 shows how a pre-instantiable service unit's readiness state is derived from operational state, presence state and administrative states of itself and administrative states of its enclosing node, service group, application and cluster. The same table

applies as is for non-pre-instantiable service units without the service unit's presence state column.

Table 3 Service Unit's Readiness State

Cluster's Administrative State	Applications's Administrative State	Service Unit's Administrative State	Service Group's Administrative State	Node's Administrative State	Nodes'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.					enabled	enabled	any	stopping
All other combinations of locked/locked-instantiation/unlocked/shutting-down, enabled/disabled and any presence state.								out-of-service

3.3.1.5 Service Unit's HA State per 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, which 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, this 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 there is no user 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 which 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 (refer to Section 3.4 for a description of switch-over).

Note: It is important to note that 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; however, in the remainder of the document, the usage of the terminology “active or standby service units” will be deemed legal when the context makes it obvious, instead of explicitly mentioning for which service instance(s) the service unit has been assigned a particular HA state. This 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, etc.) 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 3.7 on page 69 describes how these assignments are performed for the various redundancy models.

While 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.

3.3.2 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: There is no restriction in the applicability of various states of a component and their values described in the following subsections to proxied components. However, if a proxied component's status changes to unproxied (typically when it is no longer proxied), the values for various states of a proxied component reflect the last know value of the corresponding states before it became unproxied.

3.3.2.1 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**

If, after all possible retries, a component cannot be instantiated, the component's presence state is set to instantiation-failed. If components were already instantiated or instantiating 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.

If the Availability Management Framework fails to terminate a component, the component's presence state is set to termination-failed (refer to Section 4.6).

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 (see Section 3.10 for details) or in the context of a restart administrative operation (see Section 6.4.6 for details). Restarting a component means first terminating it and then instantiating it again (see Section 3.12.1.2). Two different action 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 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 a configuration attribute of each component.(Referred to as *disableRestart* for convenience in rest of the document.)

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

Table 4 shows the possible presence states of the components of a service unit, for each valid presence state of the service unit:

.

Table 4 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

Table 4 Presence State of Components of a Service Unit

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

3.3.2.2 Operational State

The **operational state** of a component refers to the ITU X.731 state management model ([5]). It is used by the Availability Management Framework to determine whether a component is capable of taking component service instance assignments. The operational state indicates whether the components within the service unit are operable or not. 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 in order 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 with the API function *saAmfComponentErrorReport()*. Such an error can be reported by the component itself, by another component, or by a monitoring facility (see *saAmfPmStart()*).
- 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.
- 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 4.4).
- If the component is SA-aware, and it unexpectedly unregisters with the Availability Management Framework. (see Section 6.1.1).
- 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 more details about recovery operations, refer to Section 3.12.1.

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 and hence the component's operational state 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 via the Availability Management Framework API.

3.3.2.3 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. This state is the only state 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 component's readiness state 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 component's readiness state 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 may not have been assigned any component service instance yet.
- **stopping:** The component's readiness state 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 5 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

3.3.2.4 Component's HA State 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 sub-actions on the components contained in the service unit, 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 operations, 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 there is no user 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, which 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 (refer to Section 3.4 for a description of switch-over).

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 appearance 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 active appears in a row at the component or component service instance level, the state of the same row at the service unit or service instance level is active. If in a row at the component or component service instance level there is no active but 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 6 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 while the service unit's HA state is active. The second row is specific for a transition of the service unit's HA state 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 which is characterized by that component service instance.

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 ser-

vice instance, if requested by the Availability Management Framework. How the standby component prepares itself is very dependent on its implementation and may involve, for example, actions such as sharing access to checkpointed data with the active component.

In switch-over situations (see Section 3.4) 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 a state where it 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 taking messages from that queue. Work related to that component service instance, which is already in progress inside the component, should be checkpointed so that it can be completed later on by the other component, which is taking over. If the component or the way it interacts with its clients does not support checkpointing of on-going work, the work needs to either be completed immediately or an indication 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, which will take over the activity. Depending on the component's implementation, 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 7.4.7 on page 242), 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 to the Availability Management Framework through the *saAmfCSIQuiescingComplete()* function call. The invocation of the *saAmfCSIQuiescingComplete()* function implicitly transitions the HA state of the component from quiescing to quiesced for that component service instance.

When assigning the active HA state to a service unit for a particular service instance, the Availability Management Framework:

- invokes the *saAmfCSISetCallback()* callback of all SA-aware components for themselves,
- invokes the *saAmfCSISetCallback()* callback of their proxy components for all proxied components. If the proxied component is a non-pre-instantiable compo-

nent and is not already instantiated, the proxy instantiates the proxied component as part of performing the component service instance assignment,

- runs the INSTANTIATE command for non-proxied, non-SA-aware components.

When assigning an HA state other than active to a service unit for a particular service instance, the Availability Management Framework:

- invokes the *saAmfCSISetCallback()* callback of SA-aware components for themselves,
- 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,
- runs the TERMINATE command for non-proxied, non-SA-aware components.

When removing a service instance assignment from a service unit, the Availability Management Framework:

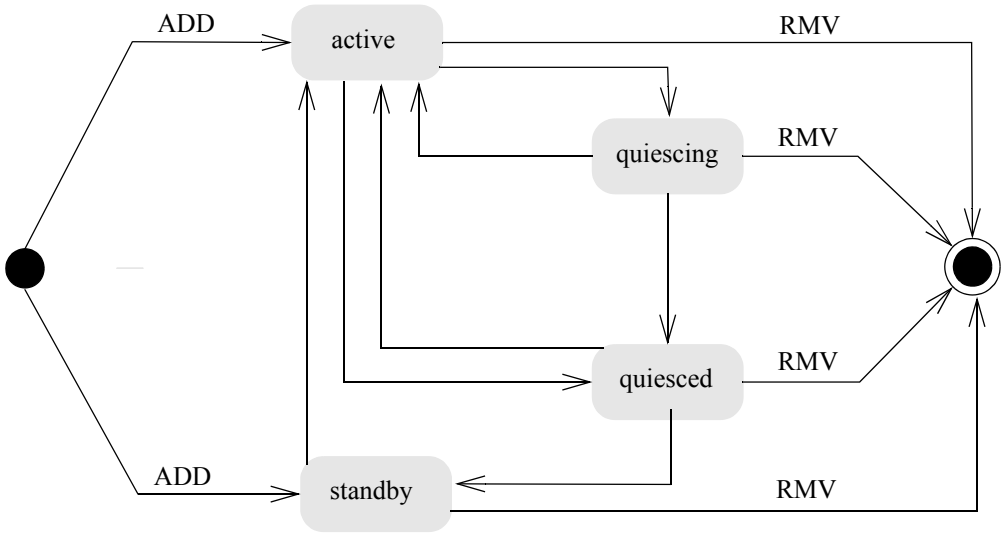
- invokes the *saAmfCSIRemoveCallback()* callback of SA-aware components for themselves,
- invokes the *saAmfCSIRemoveCallback()* callback of their proxy components for all proxied components. The proxy component terminates its non-pre-instantiable proxied components as part of removing the component service instance assignment,
- runs the TERMINATE command for non-proxied, non-SA-aware components.

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

During an individual component restart because of a fault encountered by the component, the component remains enabled. Its readiness state may or may not change according to changes in its presence state as described 3.3.2.1, which determines AMF's actions regarding the CSI assignments to the component.

The state diagram for a component service instance is shown in Figure 4 below.

Figure 4 State Diagram of the HA State of an SA-Aware Component for a Component Service Instance



ADD Transitions: saAmfCSISetCallback(SA_AMF_CSI_ADD_ONE)

RMV Transitions: saAmfCSIRemoveCallback(),

saAmfTerminateCallback(), CLEANUP

Other Transitions: saAmfCSISetCallback(SA_AMF_CSI_TARGET_*)

Table 7 below 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 the application developer.

Table 7 Application Developer View for Pre-Instantiable Components

Component's Readiness State	In-Service	Stopping	Out-Of-Service
HA State of a Component for a Component Service Instance	active standby quiescing quiesced	standby quiescing quiesced	[no HA state]

Table 8 below shows combinations of the readiness state and the HA state for non-pre-instantiable components for the component service instances that are exposed to the application developer.

Table 8 Application Developer View for Non-Pre-Instantiable Components

Component's Readiness State	In-Service	Out-Of-Service
HA State of a Component for a Component Service Instance	active or no HA state	[no HA state]

3.3.3 Service Instance States

3.3.3.1 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 via the Availability Management Framework API.

The administrative state of a service instance is persistent 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 it may not be terminated and made non-instantiable as other logical entities may be.

3.3.3.2 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 there is no service unit having 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, defined in the redundancy model of the corresponding service group (see Section 3.7) 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, defined in the redundancy model of the corresponding service group (see Section 3.7).
- **partially-assigned:** A configured service instance which 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 3.7):

Table 9 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 in the service group
N-Way Active	As configured in the service group	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 via 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 Section 9).

3.3.4 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 that comprises this component service instance.

3.3.5 Service Group States

The only state defined by the Availability Management Framework for service groups is the **administrative state.** It is an extension of the administrative state proposed by the ITU X.731 state management model ([3]) which can be manipulated by the system administrator. 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 3.3.1.4.

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

The administrative state of a service group is not directly exposed to components via the Availability Management Framework, but rather only indirectly, since the service unit's readiness state 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.

3.3.6 Node States

3.3.6.1 Administrative State

The administrative state of a node is an extension of the administrative state proposed by the ITU X.731 state management model ([3]). 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 3.3.1.4.

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

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

3.3.6.2 Operational State

The **operational state** of the node refers to the ITU X.731 state management model ([5]). It is used by the Availability Management Framework to determine whether a service unit within the node is capable of taking service instance assignments. The operational state of the node indicates whether the service units within the node are operable or not.

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 3.12.1.4).

- **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).

A node's operational state 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. It is again enabled after a successful repair. This is done 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 3.3.1.4. The operational state of a node is valid even after a node left the membership, since 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 via the Availability Management Framework, but rather only indirectly, since the service unit's readiness state has an impact on component service instance assignments.

When a node joins the cluster for the first time, the operational state of the node is enabled. The node remains enabled until a recovery within its scope is done and is reenabled when successfully repaired.

If a node is enabled and is 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 is 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 3.12.1.4 on page 139) turned on unexpectedly leaves the cluster membership, the Availability Management Framework should assess the state of the node when it rejoins the cluster membership to ascertain if it needs to proceed with the planned repair action which 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 re-enabled 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.

3.3.7 Application States

The only state defined by the Availability Management Framework for an application is the administrative state. It is an extension of the administrative state proposed by the ITU X.731 state management model ([3]) which can be manipulated by the system administrator. 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 3.3.1.4.

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 via the Availability Management Framework, but rather only indirectly, since the service unit's readiness state has an impact on component service instance assignments.

3.3.8 Cluster States

The only state defined by the Availability Management Framework for a cluster is the administrative state. It is an extension of the administrative state proposed by the ITU X.731 state management model ([3]) which can be manipulated by the system administrator. 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 3.3.1.4.

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 via the Availability Management Framework, but rather only indirectly, since the service unit's readiness state has an impact on component service instance assignments.

3.3.9 Summary of States Supported for the Logical Entities

Table 10 summarizes the states that the Availability Management Framework supports for the logical entities of the system model.

Table 10 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, presence
Component	Operational, 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 to provide service or not depends on the service unit's administrative state and on the administrative states of its containing node, service group, application, and the cluster. The corresponding rules are given in Section 3.3.1.2.

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 the these states as explained below:

- A service unit failure can lead to its presence state changing to uninstantiated and the operational state changing to disabled. This is an example of an event that changes both the operational and presence states.
- When a service unit is administratively terminated (refer to Section 7.4.4 on page 236), its presence state changes to uninstantiated, its administrative state changes to locked-instantiation, but its operational state remains unchanged. Thus, this event changes both administrative and presence states but not the operational state.

3.4 Fail-Over and Switch-Over

The terms fail-over and switch-over are used to designate two scenarios where 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 through the execution of CLEANUP commands for non-proxied components (see Section 4.6) or the invocation of *saAmfProxiedComponentCleanupCallback()* callbacks (see Section 6.9.3) for proxied components.

The term switch-over is used to designate circumstances where 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 3.12). 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.

Similarly, the terms fail-over and switch-over are also used to designate situations where 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 while a fail-over operation is the consequence of a failure recovery. As described in Section 3.12, 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.

3.5 Possible Combinations of States for Service Units

3.5.1 Combined States for Pre-Instantiable Service Units

Table 11 and Figure 5 show the possible combinations of states for these service units,

Table 11 Combined States for Pre-Instantiable Service Units

Service Unit is	Operational	Presence	Readiness	HA
locked	enabled	instantiated	out-of-service	[no HA state]
locked	enabled	uninstantiated	out-of-service	[no HA state]
locked	disabled	uninstantiated terminating instantiation-failed termination-failed	out-of-service	[no HA state]
unlocked	disabled	uninstantiated terminating instantiation-failed termination-failed	out-of-service	[no HA state]
shutting-down	enabled	instantiated	stopping	quiescing quiesced
unlocked	enabled	instantiated	in-service	any
unlocked	enabled	instantiated	out-of-service	[no HA state]
locked-instantiation	disabled	uninstantiated terminating instantiation-failed termination-failed	out-of-service	[no HA state]
locked-instantiation	enabled	uninstantiated	out-of-service	[no HA state]

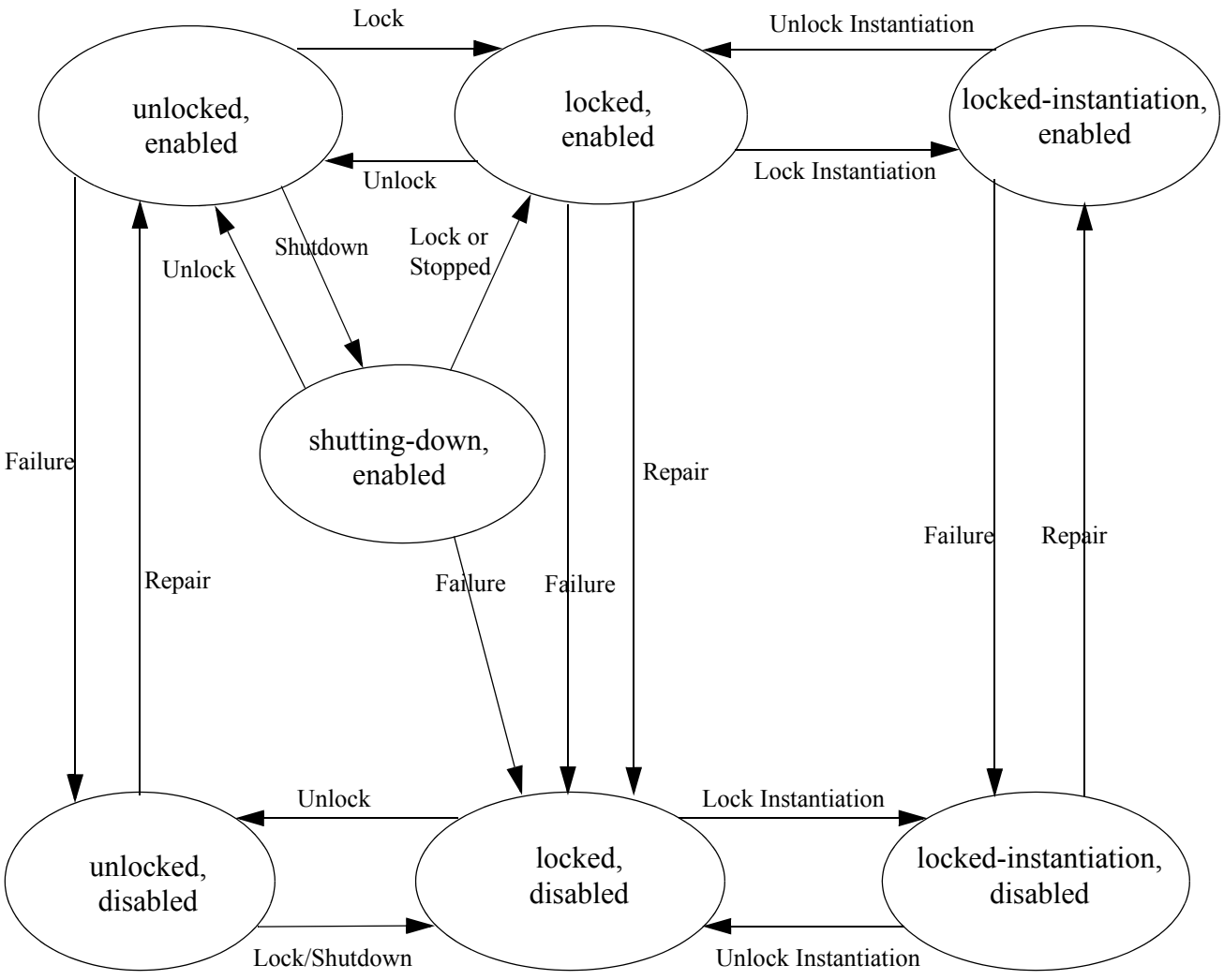
Reasons for a service unit to move from one combination of states to another:

- lock, lock-instantiation, shutdown, unlock-instantiation or unlock operation,
- failure of a contained component, which escalates to disabling the containing service unit, and, thus, cleaning up and uninstantiating the service unit,
- repair of a failed service unit (instantiating the service unit or rebooting the node, or with an SA_AMF_ADMIN_REPAIRED administrative operation),

- all contained components leaving the SA_AMF_HA_QUIESCING state for all their component service instances (labeled with "Stopped" in the diagram below)

Some of the important state transitions for a pre-instantiable component are shown below in Figure 5:

Figure 5 State Transitions for Pre-Instantiable Service Units



3.5.2 Combined States for Non-Pre-Instantiable Service Units

Table 12 and Figure 6 show the possible combinations of states for these service units.

Table 12 Combined States for Non-Pre-Instantiable Service Units

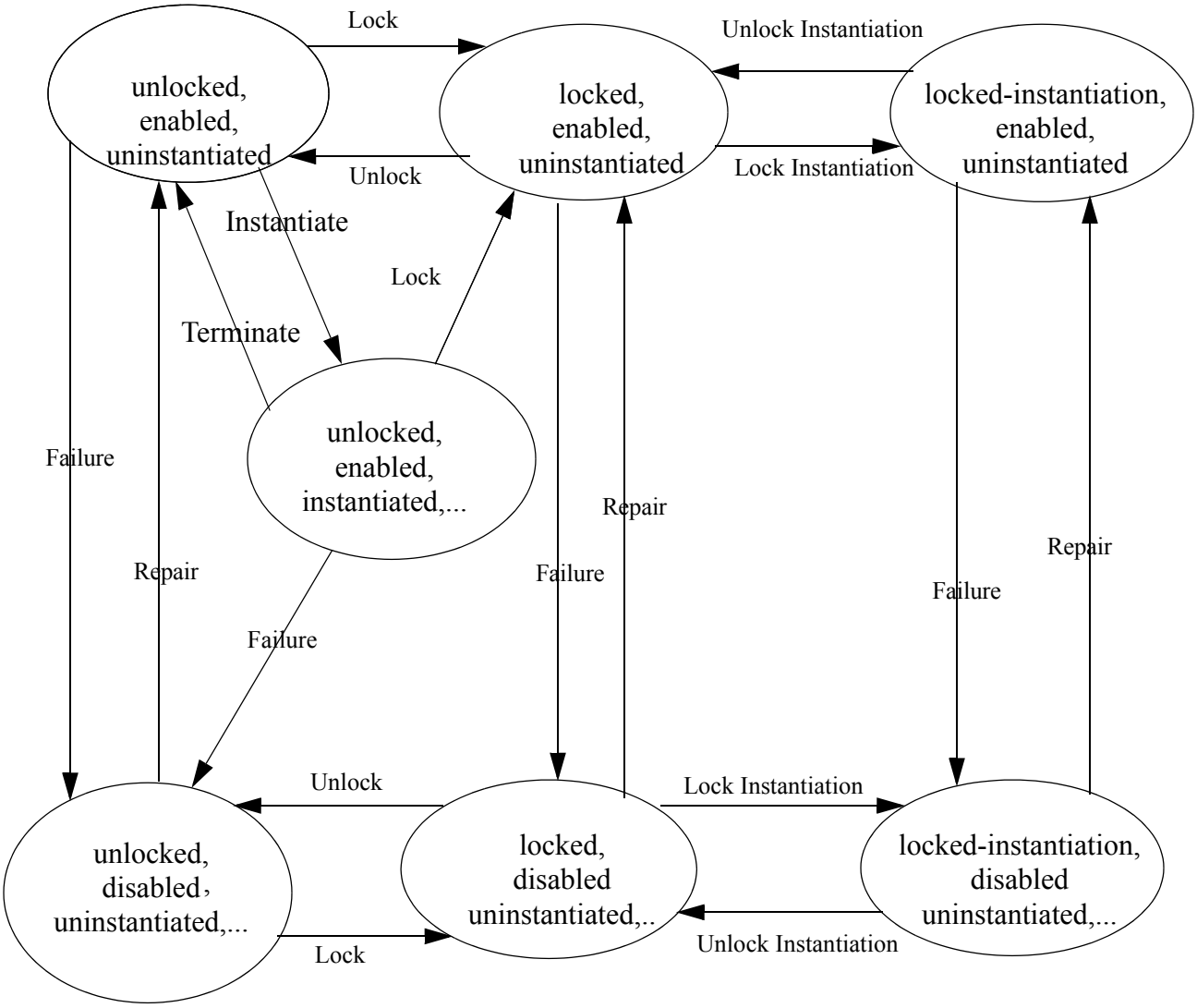
Service Unit is	Operational	Presence	Readiness	HA
locked	enabled	uninstantiated	out-of-service	[no HA state]
locked	disabled	uninstantiated instantiation failed termination failed	out-of-service	[no HA state]
unlocked	disabled	uninstantiated instantiation failed termination failed	out-of-service	[no HA state]
unlocked	enabled	uninstantiated	in-service	active
unlocked	enabled	instantiated instantiating restarting	in-service	active
unlocked	enabled	uninstantiated instantiated instantiating restarting	out-of-service	[no HA state]
locked-instantiation	disabled	uninstantiated instantiation failed termination failed	out-of-service	[no HA state]
locked-instantiation	enabled	uninstantiated	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 operation
- failure of a contained component, 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
- service unit instantiated by the Availability Management Framework

Some of the important state transitions for a pre-instantiable component are shown below in Figure 6:

Figure 6 State Transitions for Non-Pre-Instantiable Service Units



3.6 Component Capability Model

To accommodate possible simplifications in component development, whereby components may implement only restricted capabilities, the Application Interface Specification defines the Component Capability Model, which consists of the capabilities presented below. Note that the letters x and y in the name of a component capability only indicate multiplicity of numbers of active or standby component service instances. The precise values of the maximum number of active and standby CSIs will be defined in the component configuration.

- **x_active_and_y_standby:** 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:** 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:** 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:** 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:** 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:** 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:** 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.

3.7 Service Group Redundancy Model

A service group has associated with it, by configuration, a **service group redundancy model**. 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 assigning 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 shutdown administrative operations.

In the following descriptions, a service unit assigned the quiescing HA state for a service instance must be accounted as if the same service unit is assigned the active HA state for that same service instance.

The transient quiesced HA state is peculiar, as during the switch-over operation, a service unit assigned the quiesced HA state for a service instance must be accounted as either being assigned the active or standby HA state for that service instance.

In the following descriptions, 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 following descriptions, 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 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 3.7.7 on page 128 describes the effect of administrative operations on the redundancy models.

3.7.1 Common Characteristics

Note: In the following description, several ordered lists, like ordered list of service units or ordered list of service instances, are used. The order of the elements in the

list is based on the relative importance of these elements. The term rank is used as a synonym to this order. Similarly, ranked list is also used as a synonym to ordered list.

3.7.1.1 Common Definitions

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

- **Instantiable service units**: These service units have the following characteristics:
  (*) Configured in the Availability Management Framework.
  (*) Contained in a node that is currently a member of the cluster whose operational state is enabled.
  (*) The service unit's presence state is uninstantiated and its operational state is enabled.
- **in-service service units**: These are the service units that have a readiness state of either in-service or stopping.
- **Instantiated service units**: Instantiated service units: In the context of this discussion, these are service units with the presence state of either instantiated, or 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 3.7.2.2. 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. If the Availability Management Framework needs to choose a service unit for assignment from the instantiated service units list, 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 named "non-instantiated spare service units".
- **Ordered list of service units for a service group**: For each service group, there is an ordered list of service units, which defines the rank of the service unit within the service group. The length 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 also can be used to determine the order in which a service unit is selected for SI assign-

ments, 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 optimal assignments if the preferred number of its service units can actually be instantiated during the cluster start-up. In the event that a service unit or a node fails to instantiate during cluster start-up or is administratively taken out of service, a reduction procedure is described in most of the redundancy models. The Availability Management Framework uses this reduction procedure to compute less optimal assignments before actually starting to assign the service instances.
- **No spare HA state**: As spare service units have no SI assigned to them, there is no "spare" HA state for service units, respectively components on behalf of service instance, respectively component service instances. Hence, protection groups do not contain components of the spare service units; thus, no change needs to be tracked.
- **Auto-adjust option**: This notion 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. If the auto-adjust option is not set, even when a higher ranked service unit becomes eligible to take assignments (for example after a new node joining the cluster), the HA assignments to service units are kept unchanged. Refer to Section 3.7.1.2 for details when the auto-adjust option is initiated.

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 lowest the integer value, the highest 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, which is currently active for that service instance, fails.
  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 Availability Management Framework will use this ranking to choose SIs to either support with less than the desired redundancy or drop them completely, if the set of instantiated service units does not allow full support of all SIs.
- **Redundancy level of a Service Instance**: This is the number of service units being assigned an HA state for this service instance.

While most redundancy models are applicable to service groups containing non-pre-instantiable service units (see Table 13, Section 3.8), the descriptions provided in the following sections only apply 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 attached to service groups with non-pre-instantiable service units:

- no spare service units,
- no standby service units,
- 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.

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

If a service group is configured with the auto-adjust option set, then the Availability Management Framework should attempt to return the service group's assignments back to the most preferred assignments as defined in Section 3.7.1.1, as soon as possible. In general, the need for auto-adjustment for a service group arises when one of the following happens:

- 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.
- A locked service instance, configured for the service group becomes unlocked.

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, a service group-level configuration item named "probation period" has been introduced. When

a service unit becomes available for auto-adjustment after a repair/recovery operation, the service unit enters its probation period, and it cannot thereby be used for auto-adjustment while in its probation period. Note that the service group can be auto-adjusted using other service unit's but auto-adjustment cannot use the service units in their 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 probation period of a service unit elapses, the Availability Management Framework initiates the auto-adjust procedure for the corresponding service group.

By configuring the probation period appropriately, the administrator can make sure that the Availability Management Framework does not run into undesired situations such as toggling the active service units due to, for example, intermittent failures of a service unit and inadequate repair operations.

3.7.2 2N Redundancy Model

3.7.2.1 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, there is no redundancy on 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 viewpoint, 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 3.6 on page 68 can participate in the 2N redundancy model.

Examples of a service group with a 2N redundancy model are presented in Section 3.7.2.4 on page 76.

3.7.2.2 Configuration

- **Ordered list of service units for a service group**: This parameter is described in Section 3.7.1.1.
  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 are always instantiated, if possible. 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, then there will be some instantiated spare service units in the service group. 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 3.7.1.1 on page 71. Section 3.7.2.3.3 on page 75 discusses how this option is handled in this redundancy model.
  Default value: no auto-adjust

3.7.2.3 SI Assignments and Failure Handling

3.7.2.3.1 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.

3.7.2.3.2 Failure of the Standby Service Unit

When a standby service unit fails over, 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.

3.7.2.3.3 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.

3.7.2.3.4 Cluster Startup

Because the cluster startup is a rare event, its latency may not be as critical as other failure recovery events such as a service unit fail-over. Moreover, it is very important to start a cluster in an orderly fashion such that the initial runtime status of the entities under the control of the Availability Management Framework 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. 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.

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

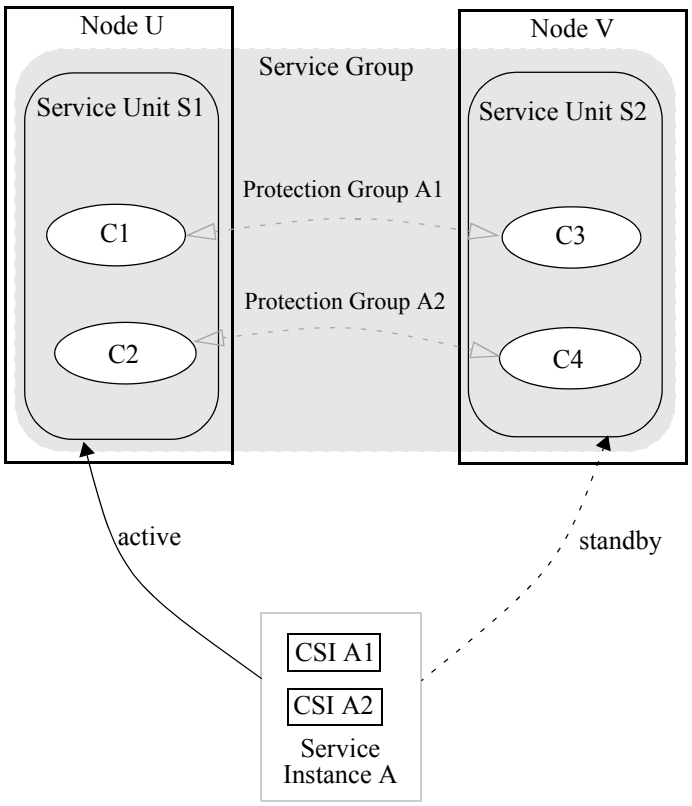
The ordered list of service units will be used for the following purposes:

- To decide when to instantiate a service unit from the list: If, at a given time, the number of instantiated service units is less than the preferred number of in-service service units specified in the configuration, the non-instantiated service units with highest ranks in the service unit list will be instantiated until both numbers are equal. If the preferred number of in-service service units cannot be instantiated due to a shortage of instantiable service units, then the service group will be only partially supported.
- To select which of the instantiated service units will have active and standby assignments: When there is no active or standby assignments for a service group and several service units are instantiated at the same time (for example during the cluster startup or when multiple nodes join the cluster at the same time), then, for each SI, the Availability Management Framework will assign the service unit with highest rank in the list the active HA state for this SI and the second highest ranked service unit the standby HA state for this SI.

3.7.2.4 Examples

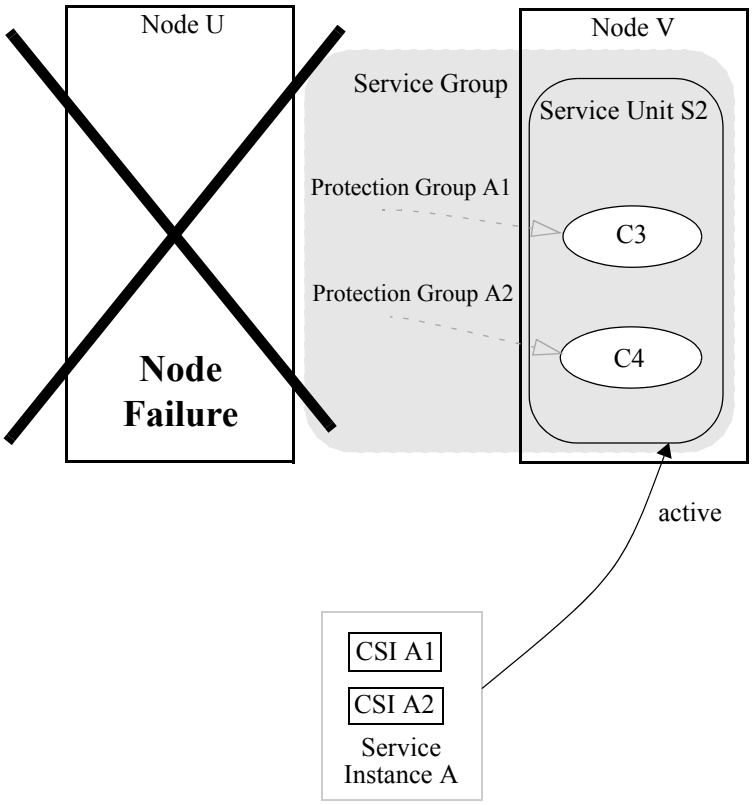
In the following example, it is assumed that the number of preferred in-service service units is set to 2.

Figure 7 Example of 2N Redundancy Model: Two Service Units on Different Nodes



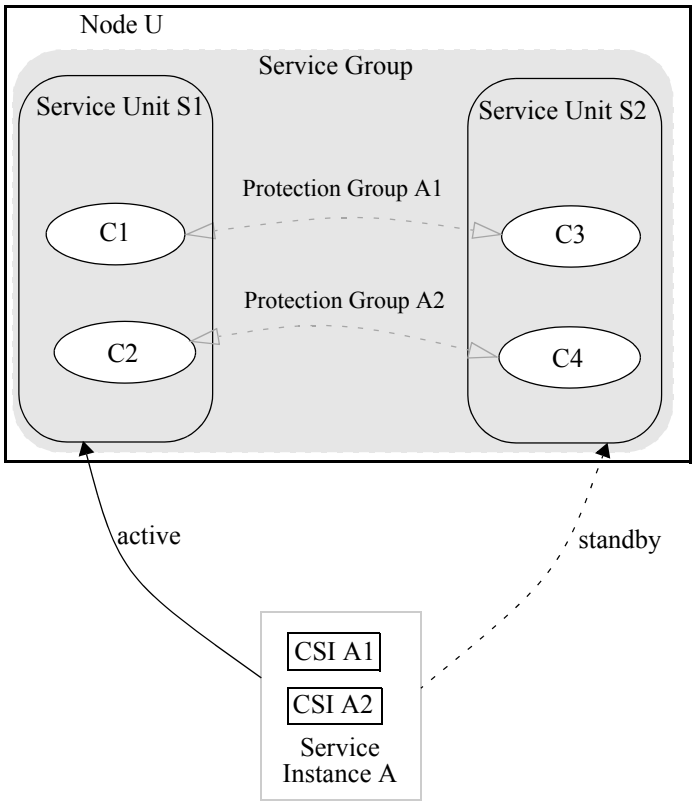
After a fault that disables Node U, Service Unit S2 on Node V will be assigned to be active for Service Instance A, as shown in Figure 8 on page 78.

Figure 8 **Example of 2N Redundancy Model. Two Service Units on Different Nodes Where a Fault Has Occurred**



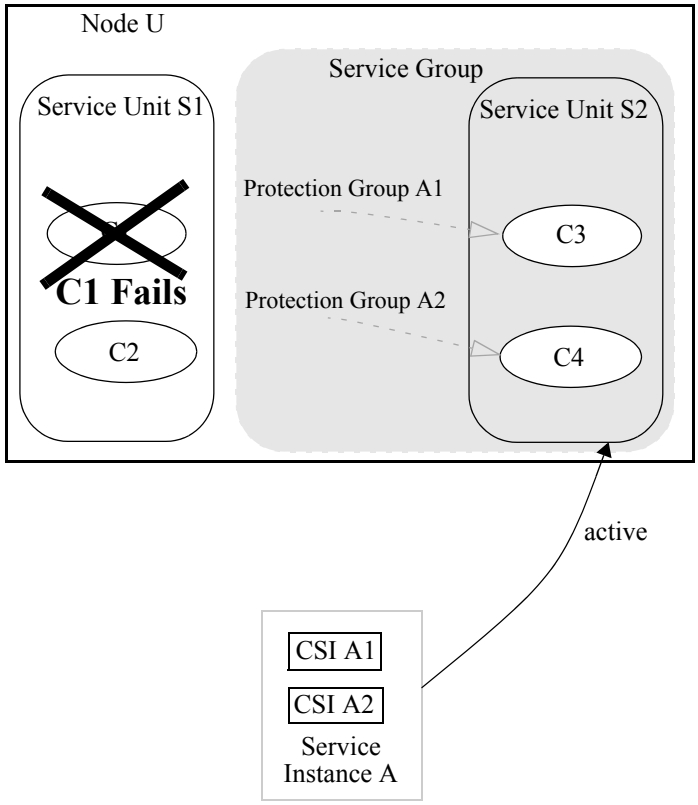
The two service units may even reside on the same node, as shown in Figure 9 on page 79, which allows one to implement software redundancy with two instances of the application running on the same node.

Figure 9 Example of 2N Redundancy Model: Two Service Units on the Same Node



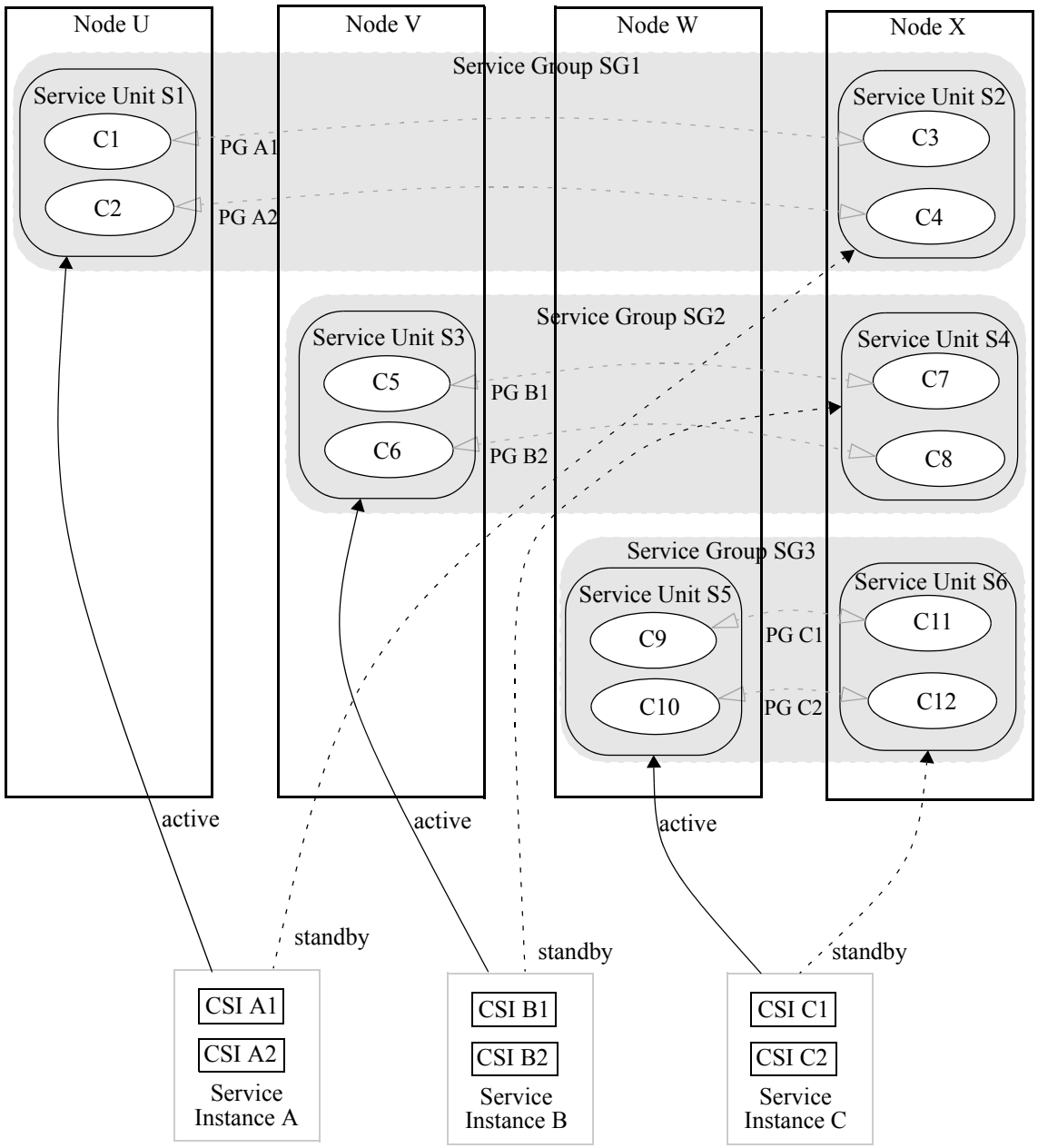
As shown in Figure 10 on page 80, after a fault that disables component C1 within service unit S1, service unit S2 is assigned to be active for service instance A. Note that a fault affecting any component within a service unit that cannot be recovered by restarting the affected component, causes the 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 10 Example of 2N Redundancy Model: Two Service Units on the Same Node, Where a Fault Has Occurred



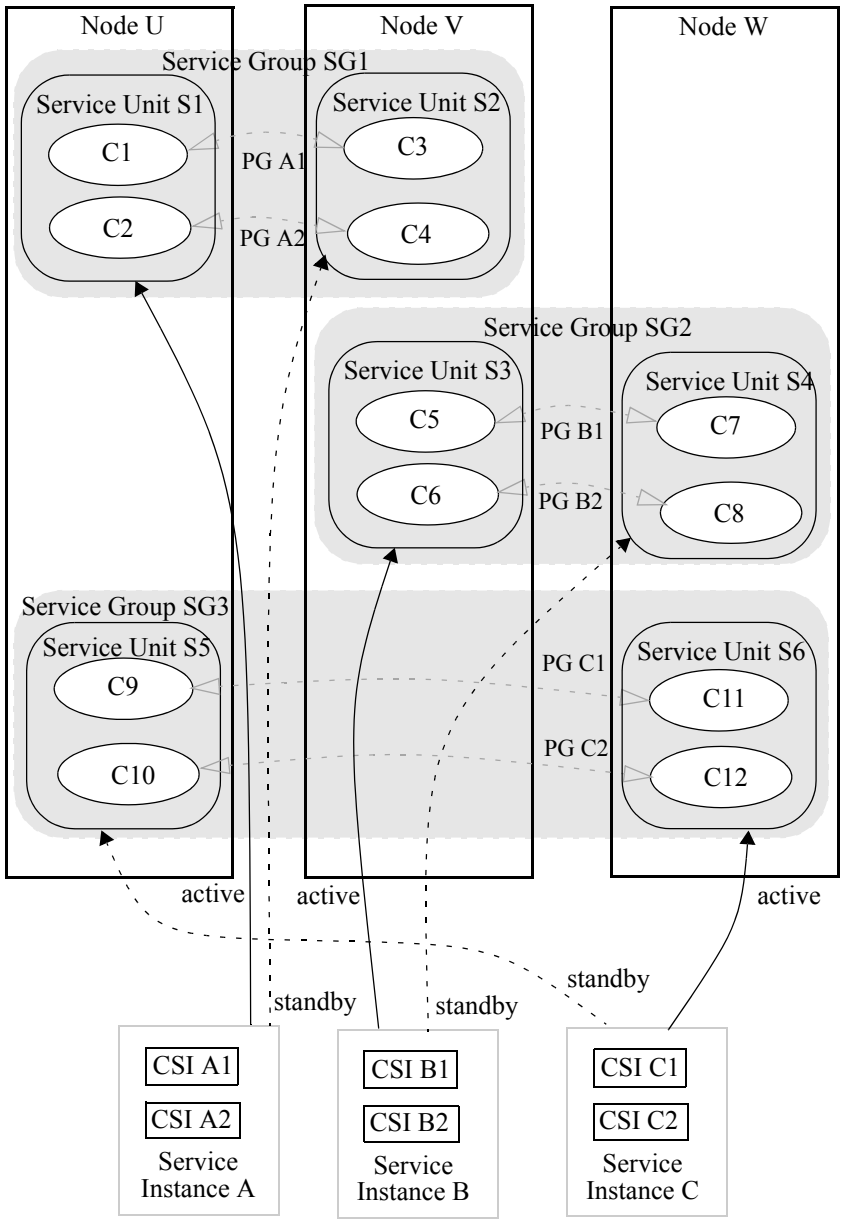
As shown in the next figure, the 2N service group redundancy model can support N+1 strategies at the node level, as shown below. Node X supports standby service units for several service groups. If one of the other nodes fails, the corresponding service unit on Node X, and its components, will be reassigned 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 11 **Example of 2N Redundancy Model, Where a Single Node Provides Standby Service Units for Several Service Groups**



As Figure 12 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.

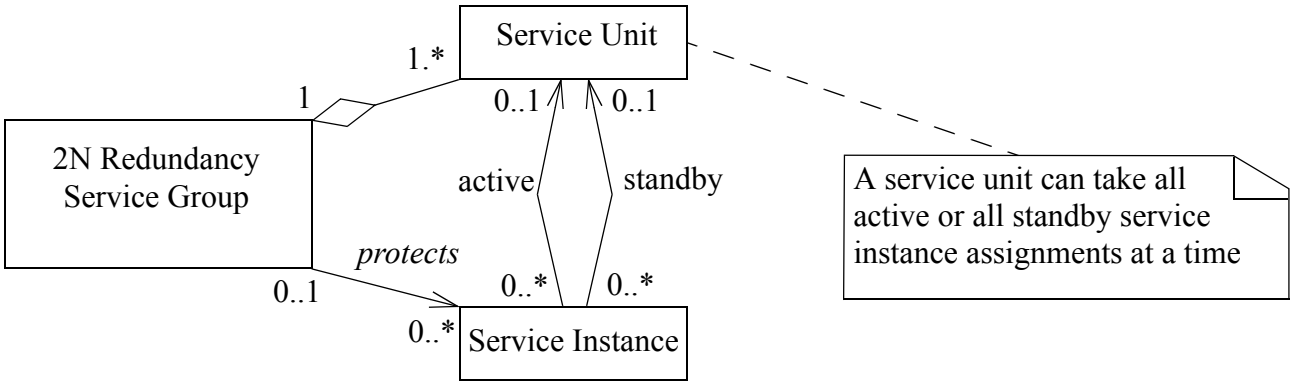
Figure 12 Example of 2N Redundancy Model, Where Nodes Support Some Service Units That Are Active for Their Service Instances and Other Service Units That Are Standby for Their Service Instances



3.7.2.5 UML Diagram of the 2N Redundancy Model

The 2N redundancy model is represented by the UML diagram shown in the following figure.

Figure 13 UML Diagram for 2N Redundancy Model



3.7.3 N+M Redundancy Model

3.7.3.1 Basics

In the **N+M redundancy model**, the service group has N+M service units.

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, there can be several in-service service units 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 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 3.7.3.3 on page 86.

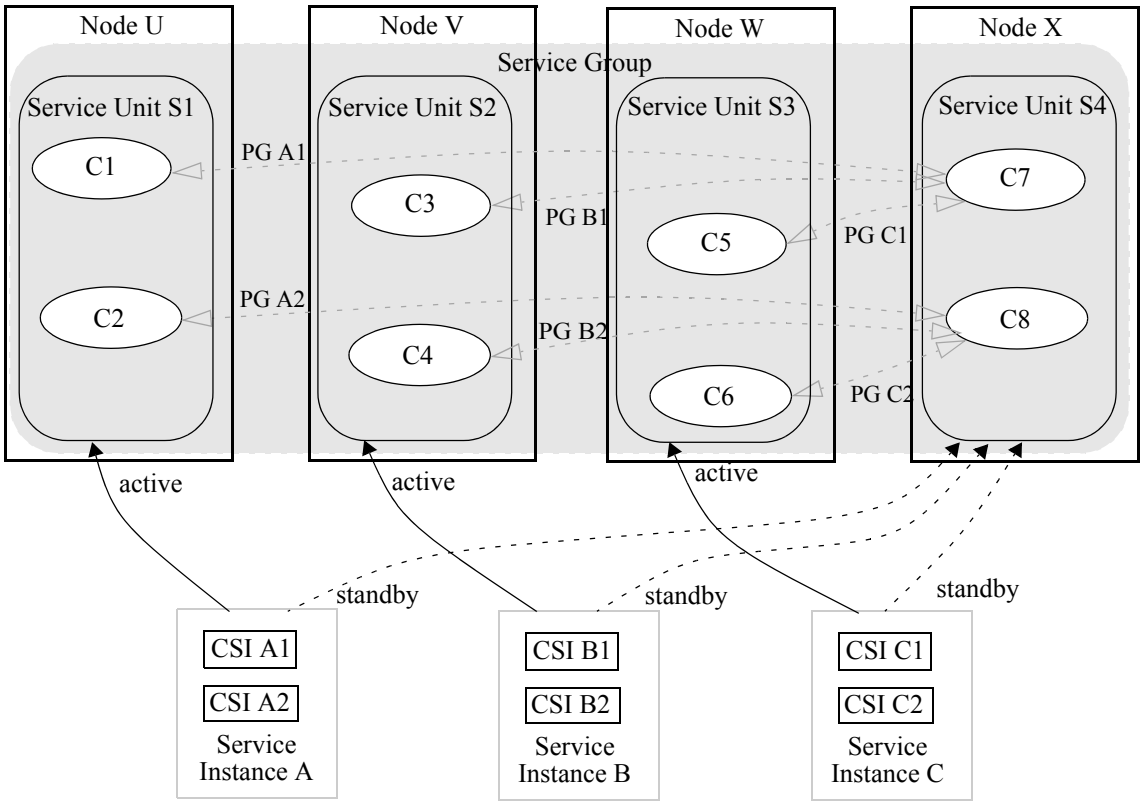
- For each SI and at any given time, there will be at most one active service unit and at most one standby service unit.
- At any given time, the Availability Management Framework should make sure that the per-SI redundancy level (one service unit assigned the active HA state and a service unit on another 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 3.7.3.3) 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, there may be cases when this may not be feasible due to the 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 (e.g., no service unit assigned standby for this SI). The service group deployer should be allowed to specify the order of importance of SIs, as discussed in Section 3.7.3.3.

Components implementing any of the capability models described in Section 3.6 on page 68, except the 1_active _or_1_standby capability model, can participate in the N+M redundancy model.

3.7.3.2 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 14 on page 85. 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.

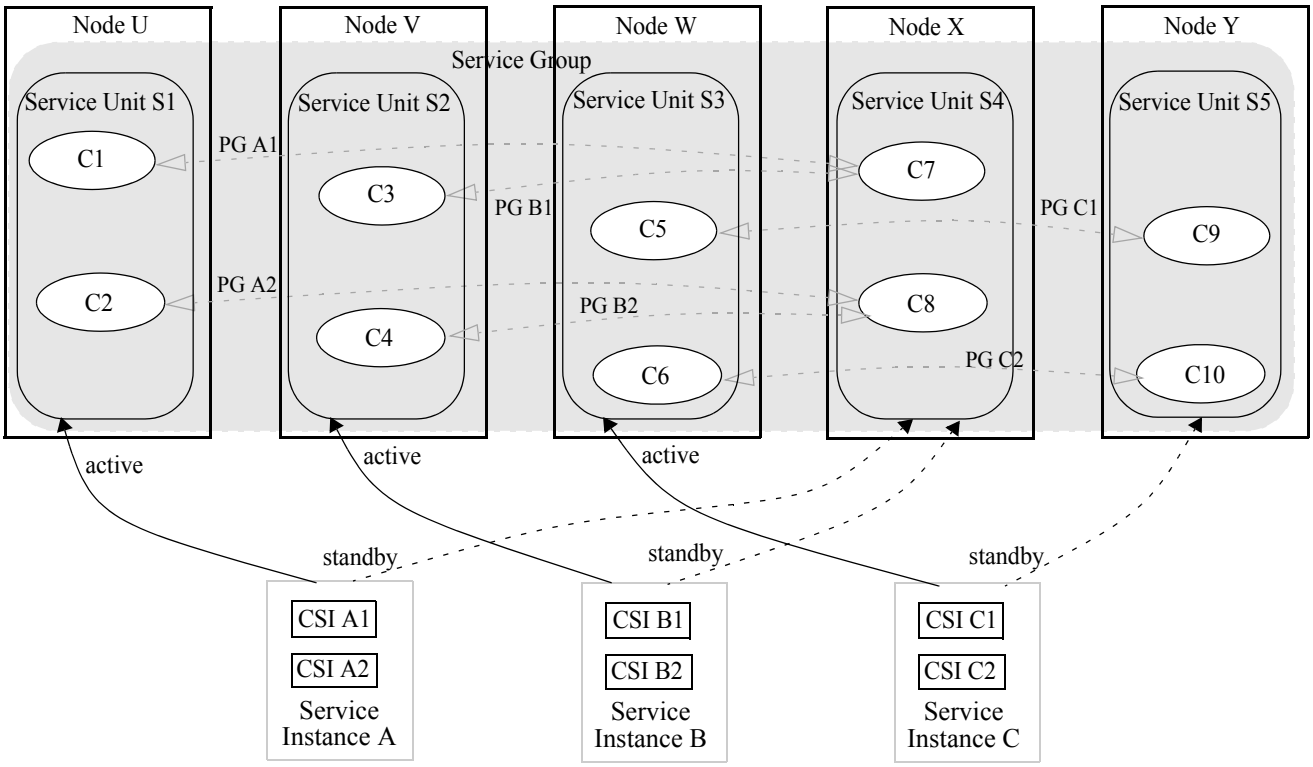
Figure 14 Example of N+1 Redundancy Model



To illustrate what happens after 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. Because, according to the redundancy model, S4 may not be assigned active for some SIs and standby for other SIs at the same time, the standby HA state for service instances A and C will be removed from S4. Note that this is also true 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. The following figure shows an example of the N+M redundancy model with N=3 and M=2.

Figure 15 Example of N+M Redundancy Model, Where N = 3 and M = 2



3.7.3.3 Configuration

- **Ordered list of service units for a service group**: This parameter is described in Section 3.7.1.1.
  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 3.7.1.1. The Availability Management Framework will use this ranking to select some SIs to support either in non-redundant mode (i.e., 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 drop them completely, if the Availability Management Framework encounters 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 in choosing some SIs to support partially or drop them. The Availability Management Framework should observe the following role assignment of the SIs: The assignment goes in an order compatible with the dependencies. If

several SIs could be assigned at the same time with respect to that criterion, the ordered list of SIs serves as a tie-breaker.
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 are always instantiated, if possible. 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. The Availability Management Framework should try to guarantee that this number of active service units exist for the service group, if 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. The Availability Management Framework should guarantee that this number of standby service units exist for the service group, if 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 for each service unit**: This indicates the maximum number of SIs that can be assigned to a service unit such 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 is not true for some service instances, then the service deployer has to approximate.
Default value: no limit (A value of 0 is used to specify this)
- **Maximum number of standby SIs for each service unit**: This indicates the maximum number of SIs that can be assigned to a service unit such 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.
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 3.7.1.1 on page 71. Section 3.7.3.6 on page 96 shows an example for handling the auto-adjust option in this redundancy model.
Default value: no auto-adjust

3.7.3.4 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. Additionally, 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 gone through a series of validations, so that when the preferred number of active (respectively standby) service units are assigned the active (respectively the standby) HA state, there will be one service unit assigned the active HA state and another one the standby HA state for each SI of the service group, without violating the load limits expressed in Section 3.7.3.3.

In case of 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 a non-redundant mode (i.e., 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).

In the rest 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 for each service unit = 3
- Maximum number of standby SIs for each 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 less service units are available due to unavailability of some nodes.

Note: This specification does not define the actual algorithm for SI assignments. A few rules are provided to guide implementations as stated above. The examples provided are only illustrative and represents one possible assignment scenario (by a particular implementation) based on the configuration specified in page 88. Implementations should design their own assignment algorithms by following the above rules".

The difficulty comes when there are not enough in-service service units to satisfy the configuration requirements. The first goal is to try to keep all SIs in the redundant mode (i.e., there is one service unit having the active HA state and another service unit having the standby HA state for each of those SIs) even at the expense of imposing maximum load on each service unit. If this goal is not attainable, then 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 may lead to a reduction in the number of standby service units. Finally, if this objective is also not attainable, then there are no choices, but 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.

3.7.3.4.1 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 desired 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

Let us 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, with every SI being in the 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 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 are available, and, for each SI, there is a service unit having 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 there are still some SIs 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

Let us 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, with every SI being in the 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, there are still not enough in-service service units to keep 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 there are still some SIs 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

Let us 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 the 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 are 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 them 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

Let us 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}

Note that, in this assignment, SI5 and SI6 are supported only in a non-redundant mode (i.e., 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).

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 there are service units being 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 a 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).

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

Assignment VI: Partial Assignment with SIs Drop-Outs

Let us 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.

3.7.3.5 Examples for Service Unit Fail-Over

The Availability Management Framework reactions to failures such as node failures are implementation-dependent and are out of the scope of the specification; however, 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.

3.7.3.5.1 Handling of a Node Failure when Spare Service Units Exist

Let us assume that the cluster configuration was as follows, 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 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 the following:

- 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}

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

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

Let us assume the following is the 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. There can be a couple of different ways of restoring the standby assignments for SI3 and SI4. It depends on the Availability Management Framework implementation how to achieve this 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}

3.7.3.6 An 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 such that the service

units with the highest ranks are active and the highest ranked SIs are assigned in redundant mode (i.e., 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. Let us 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. When the auto-adjust is initiated (in a implementation-dependent way), the service group configuration should look like as follows, after the completion of the auto-adjust procedure (assuming that SU1 could be brought in-service):

- 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}

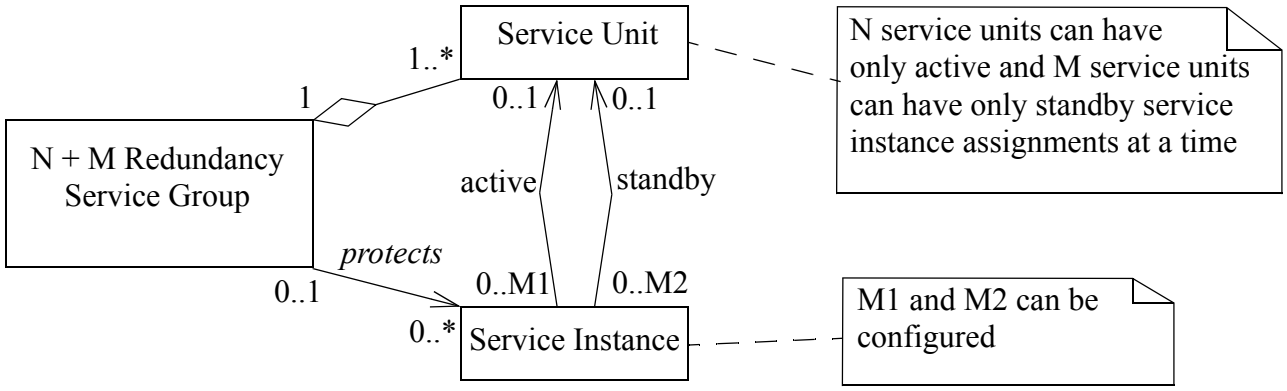
- SIs assigned to SU5 as standby = {SI3, SI4}
- SIs assigned to SU6 as standby = {SI5, SI6}

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

3.7.3.7 UML Diagram of the N+M Redundancy Model

The N+M redundancy model is represented by the UML diagram shown in the following figure.

Figure 16 UML Diagram of the N+M Redundancy Model



3.7.4 N-Way Redundancy Model

3.7.4.1 Basics

In the **N-way redundancy model**, a service group contains N service units that protect multiple service instances.

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 zero, one or multiple service units may have the standby HA state for the same SI.

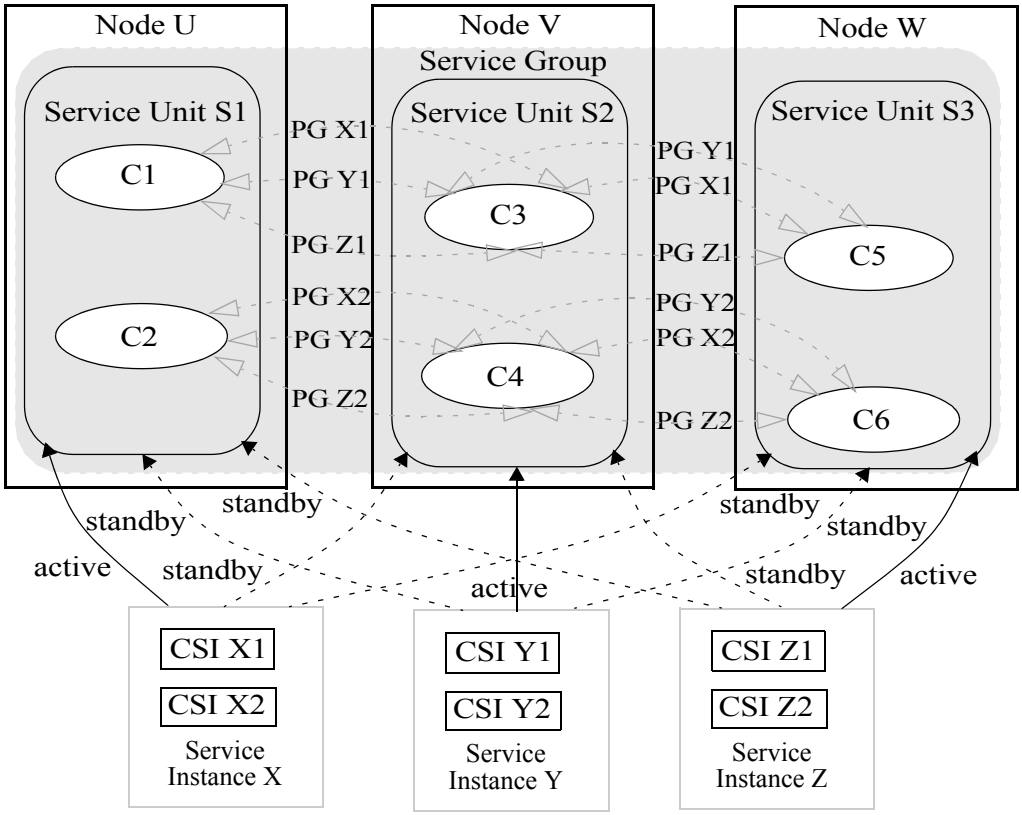
- The preferred number of standby assignments for an SI is an SI-level configuration parameter. The preferred number of standby assignments may differ for each SI.
- At any given time, there can be several service units in-service for a service group: Some have SI assignments and possibly some others 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 3.7.4.3.
- If resources allow and at any given time, the Availability Management Framework should make sure 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 the SI can be assigned to. For simplicity of the model and its implementation, it is assumed that all service units in a service group are identical. That means that any SI can be assigned to any service unit. Therefore, 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 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.

3.7.4.2 Example

Figure 17 next 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 17 Example of N-Way Redundancy Model



3.7.4.3 Configuration

- **Ordered list of service units for a service group**: This parameter is described in Section 3.7.1.1.
  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 3.7.1.1. The Availability Management Framework will use this ranking to choose SIs to support either in non-redundant mode (i.e., 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 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 the SI can be assigned to. 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 indicates the preferred number of service units that are assigned the standby HA state for this SI.
Default value: 1
- **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 are always instantiated, if possible. If the service units list for a service group includes at least two service units, then 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 indicates the preferred number of assigned service units at any time. As to be discussed in Section 3.7.4.4 on page 101, the Availability Management Framework should try to guarantee that this number of assigned service units exist for the service group, if the number of instantiated service units is large enough.
Default value: the preferred number of in-service service units.
- **Maximum number of active SIs for each service unit**: This indicates the maximum number of SIs that can be concurrently assigned to a service unit such 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.
Default value: no limit (A value of 0 is used to specify this)
- **Maximum number of standby SIs for each service unit**: This indicates the maximum number of standby SIs that can be concurrently assigned to a service unit such 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.
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 3.7.1.1 on page 71. Section 3.7.4.6 on page 106 shows an example for handling the auto-adjust option in this redundancy model.
Default value: no auto-adjust

3.7.4.4 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. Moreover, the preferred num-

ber 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 gone through 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 there is a service unit assigned active for this SI and the preferred number of standby assignments is ensured, without violating the limits expressed in Section 3.7.4.3.

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 for each service unit = 4
- Maximum number of standby SIs for each service unit = 5

Assignment I: Full Assignment with Spare Service Units

Let us 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.

3.7.4.4.1 Reduction Procedure

The difficulty comes when there are not enough in-service service units to satisfy the configuration requirements listed in the example. The first goal is to try to keep all SIs in the desired redundant mode (i.e., there is one service unit 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 reduction in the number of standby service units. The reduction is done for less important SIs first. Finally, if this objective is also not attainable, there is no choices but 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

Let us 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

Let us 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

Let us 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

Let us 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}

3.7.4.5 Failure Handling

In this section, the fail-over action initiated by a node failure is described. Let us 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).

3.7.4.6 Auto-adjust Example

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.

Let us 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 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, after completion of the auto-adjust procedure, the service group configuration looks like:

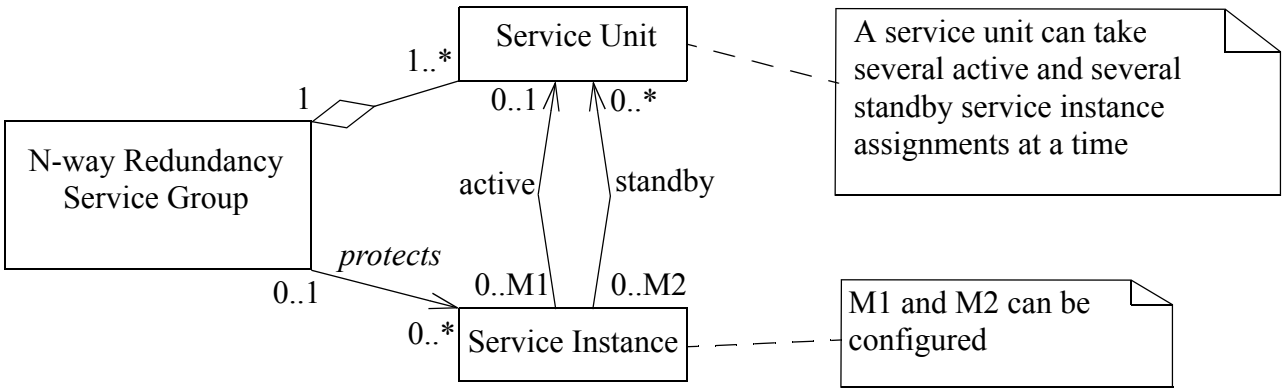
- in-service service units = {SU1, SU2, SU3, SU4}
- instantiable service units = {}
- assigned service units = {SU1, SU2, SU3, SU4}
- spare service units = {}

- 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}

3.7.4.7 UML Diagram of the N-Way Redundancy Model

The N-way redundancy model is represented by the UML diagram shown in the following figure.

Figure 18 UML Diagram of N-Way Redundancy Model



3.7.5 N-Way Active Redundancy Model

3.7.5.1 Basics

In the **N-way active redundancy model**, the service group contains N service units. 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, there may be zero, one, or multiple service units assigned the active HA state for that SI.
- Preferred number of active assignments for an SI is an SI-level configuration parameter (see Section 3.7.5.3 on page 110). The preferred number of active assignments may be different for each SI.
- At any given time, there can be several service units 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 the SI can be assigned to. 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.

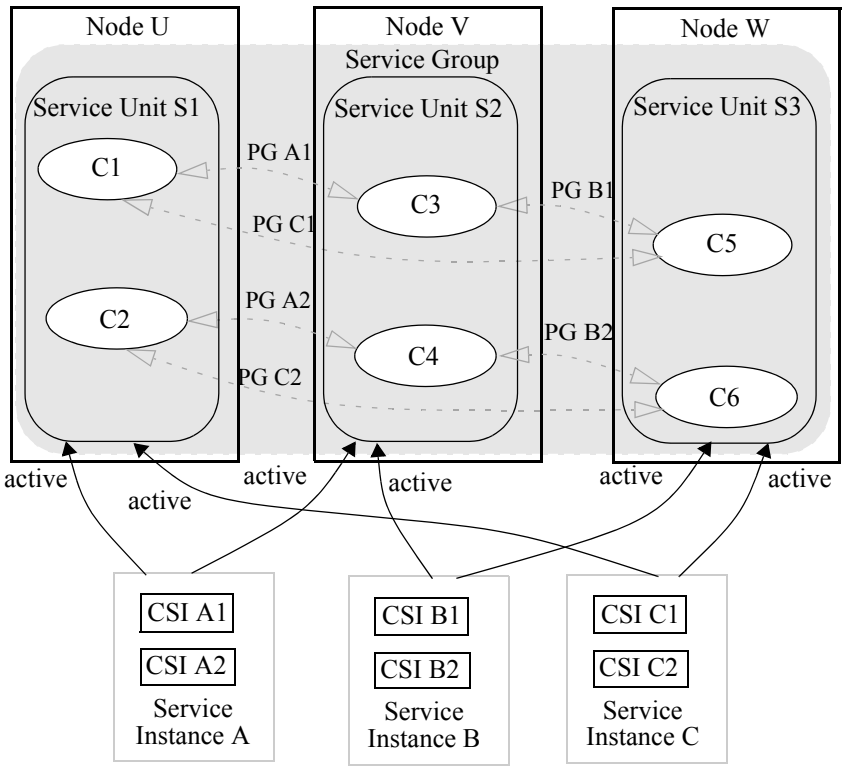
The simplest case for the N-way active redundancy model is the 2-way active redundancy model in which the service group contains two service units that are both assigned the active HA state for every service instance that they support. This is sometimes referred to as an active-active redundancy configuration.

Components implementing any of the capability models described in Section 3.6 on page 68 can participate in the N-way active redundancy model.

3.7.5.2 Example

Figure 19 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 19 Example of N-Way Active Redundancy Model



3.7.5.3 Configuration

- **Ordered list of service units for a service group**: This parameter is described in Section 3.7.1.1.
  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 3.7.1.1. The Availability Management Framework will use this ranking to choose the SIs with less redundancy (i.e., there are less than the preferred number of service units having the active HA state for them) or drop them completely, if the number of available service units are 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 the SI can be assigned to. This list must be an ordered list consisting of all service units configured for the service group. 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 assignment per SI**: This indicates the preferred number of service units being assigned the active HA state for each SI.
Default value: the preferred number of assigned service units.
- **Preferred number of in-service service units at a given time**: The Availability Management Framework should make sure that this number of service units are always instantiated, if possible.
Default value: the number of the service units configured for the service group.
- **Preferred number of assigned service units**: This indicates the preferred number of assigned service units at any time. As to be discussed later, the Availability Management Framework should try to guarantee that this number of assigned service units exist for the service group, if the number of instantiated service units is large enough.
Default value: the preferred number of in-service service units.
- **Maximum number of active SIs for each service unit**: This indicates the maximum number of SIs that can be concurrently assigned to a service unit such 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.
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 3.7.1.1 on page 71. Section 3.7.5.6 on page 119 shows an example for handling the auto-adjust option in this redundancy model.
Default value: no auto-adjust

3.7.5.4 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. Additionally, 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 the configuration allows. It is assumed that the service group configuration has gone through 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 such that 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 for each service unit = 5

Assignment I: Full Assignment with Spare

Let us 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.

3.7.5.4.1 Reduction Procedure

The difficulty comes when there are not enough in-service service units to satisfy the requirements listed in the configuration. The first goal is to try to keep all SIs in the preferred redundancy levels (i.e., 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, then 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, there is no choices but 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

Let us 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 (i.e., 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

Let us 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 service units in SIs can be summarized as follows.

The number of assignments that can be handled in this case is
number of in-service service units (i.e., 3) * maximum number of SIs for each service unit (i.e., 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 (i.e., 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

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

3.7.5.5 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 assign-

ments. 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):

3.7.5.5.1 Example for Failure Recovery

In this example, let us 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 is not “optimal” for the following two reasons:

(1) The less important SIs (i.e., SI4 and SI5) have higher levels of assignment than more important SIs (i.e., SI1, SI2, and SI3).

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

This requires failure recovery, discussed below.

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, this 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 above “optimal” solution. For example, a simple implementation that does not aim at guaranteeing "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.

3.7.5.6 Auto-adjust Example

As discussed earlier, the failure recovery should avoid undesirable situations (i.e., under-utilized 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.

So, there may be cases in which the SIs are not arranged based on their service units ordered lists. The fail-over procedure can be initiated to do one of the following rearrangements:

(1) Redistribute the SIs to service units evenly and based on the per-SI based ordering such that the SIs are distributed among all assigned service units.

(2) Rearrange the assignment such 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, let us 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 in full 6 active assignments, one thing the Availability Management Framework can do (at least) is to 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:

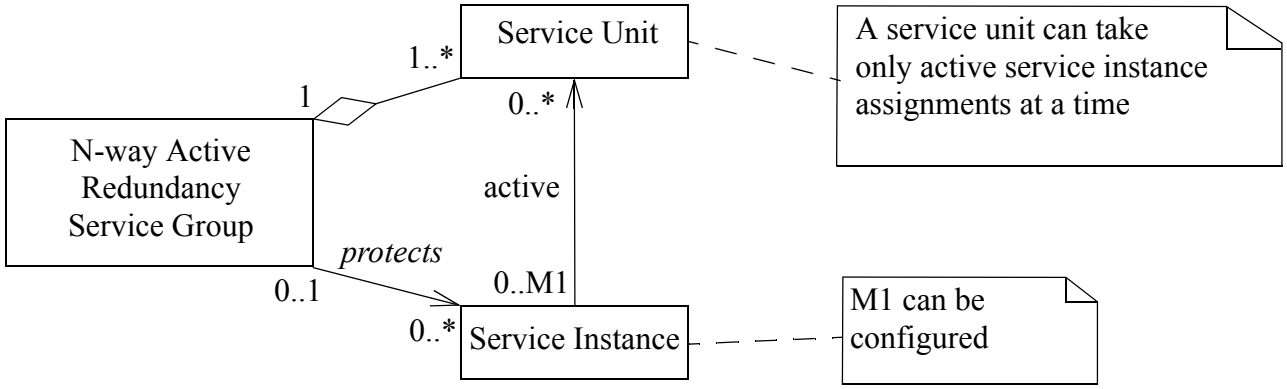
- SI1's assignments = {SU1, SU2, SU3, SU4, SU5, SU6}
- SI2's assignments = {SU2, SU3, SU4, SU5, SU6, SU7}
- SI3's assignments = {SU3, SU4, SU5, SU6, SU7, SU8}

- 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}

3.7.5.7 UML Diagram of the N-Way Active Redundancy Model

The N-way active redundancy model is represented by the UML diagram shown in the following figure.

Figure 20 UML Diagram of N-Way Active Redundancy Model



3.7.6 No Redundancy Model

3.7.6.1 Basics

In the **no redundancy** model, the service group contains one or more service units.

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 restarting a service unit, or as an escalation, by restarting the node (see Section 7.1.1 on page 229) containing the service unit.
- No two service units exist having the same SI assigned to them.

- At any given time, there can be several in-service service units 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, if 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.

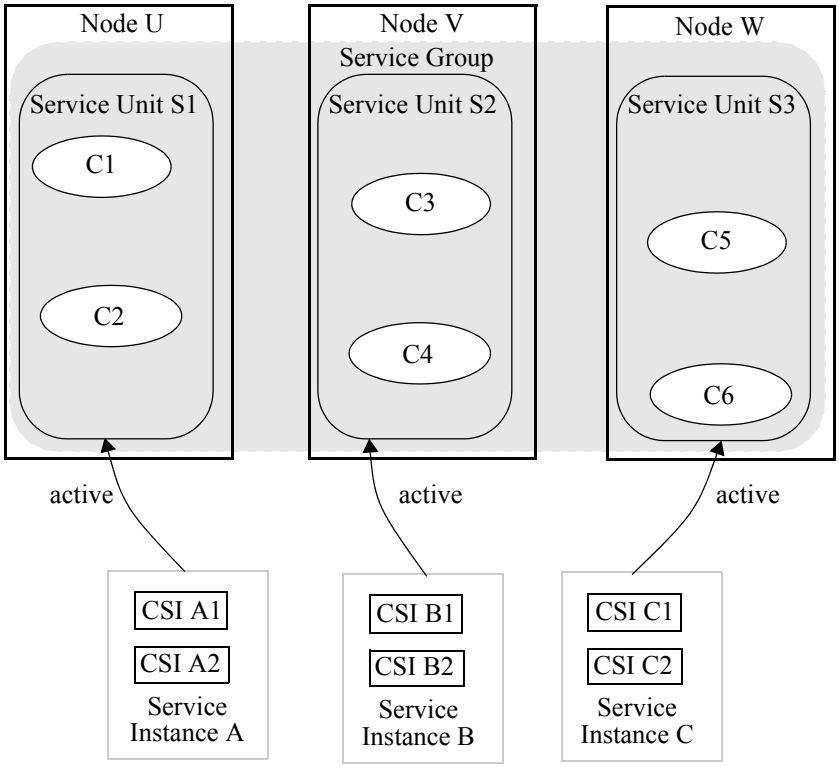
Note: To simplify the SI assignments and failure handling, it is assumed that all service units in a service group are capable of providing all SIs defined in the configuration.

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.

3.7.6.2 Example

An example of the no redundancy model is shown in the following figure.

Figure 21 Example of "No Redundancy" Redundancy Model



3.7.6.3 Configuration

- **Ordered list of service units for a service group**: This parameter is described in Section 3.7.1.1.
  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 3.7.1.1. The Availability Management Framework uses this ranking to choose the SIs to drop from assignment, if there is shortage of service units for a full support of all SIs.
  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 are always instantiated, if possible.
  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 3.7.1.1 on page 71. Section 3.7.6.6 on page 126 shows an example for handling the auto-adjust option in this redundancy model.
  Default value: no auto-adjust

Note that the preferred number of assigned service units is equal to the number of configured SIs plus one spare service unit.

3.7.6.4 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 gone through a series of validations, so that when the required number of service units are assigned, then 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

Let us 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 ordered list of service units for each SI.

3.7.6.4.1 Reduction Procedure

The first goal of the assignment procedure is to try keeping 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

Let us 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. Let us 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 = {}

3.7.6.5 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 assign the SI of the failed service unit to the selected spare service unit. If there is no spare service unit 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}

3.7.6.6 Auto-adjust Example

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 SU3 has a higher ranking for SI3 than SU4, 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}

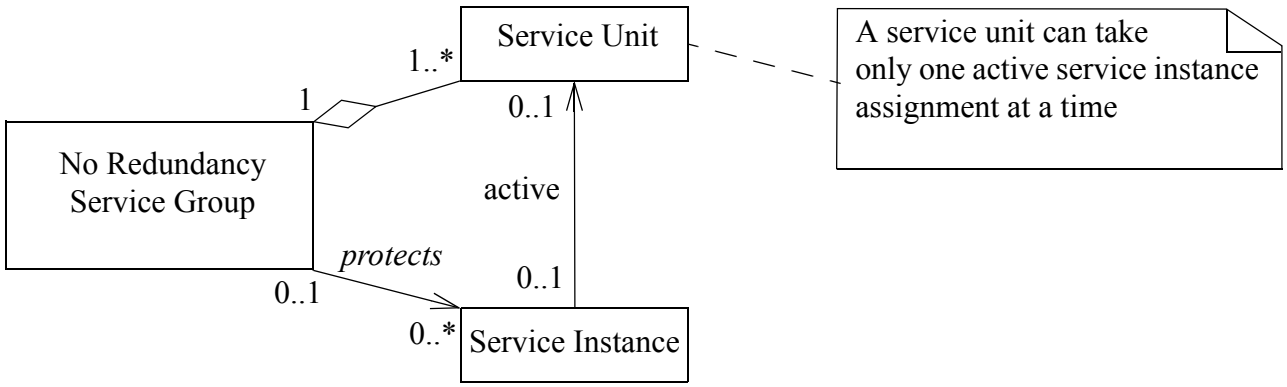
- SI1's assignments = {SU1}
- SI2's assignments = {SU2}
- SI3's assignments = {SU3}

Note that SU5 has been de-instantiated, because the number of preferred service units is 4.

3.7.6.7 UML Diagram of the No Redundancy Model

The No Redundancy redundancy model is represented by the UML diagram shown in the following figure.

Figure 22 UML Diagram of "No Redundancy" Redundancy Model



3.7.7 The Effect of Administrative Operations on Service Instance Assignments

Usually, administrative operations such as lock or unlock of a service unit or a node result in reassignments of SIs to service units. This section briefly discusses the effects for the lock/unlock operations.

The cases for other administrative operations are similar. The detailed reaction of the Availability Management Framework for each administrative operation depends on the redundancy model; however, the basic directions are given here. The details are left for the implementation.

3.7.7.1 Locking a Service Unit or a Node

Since the lock for instantiation does not effect the service instance assignment, this subsection focusses on the lock operation only.

Depending on the status of the service unit, one of the following cases can happen when locking a service unit:

(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. Transferring SI assignments from the service unit SU1 to other service units is very similar to the recovery operation performed when a service units fails. Refer to the failure handling section of the associated redundancy model for details. 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. Removing SI assignments will not trigger a termination of the service unit and the operation discussed below in case (b) is undertaken when it 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: When the service unit SU1 becomes out-of-service, if 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 3.7.1. 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: No SI reassignment or service unit instantiation is performed.

3.7.7.2 Unlocking a Service Unit, a Service Group, or a Node

After unlocking a service unit, the following cases can occur:

(a) The service unit does not belong to the set of instantiable service units. Nothing can be done in this case, and the service unit still remains out of the set of instantiated service units.

(b) The service unit belongs to the set of instantiable service units, but it is not instantiated. 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 below 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 the current assignments, some SIs may be assigned to the service unit.

3.8 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 13.

Table 13 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 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.

3.9 Dependencies Among SIs, Component Service Instances, and Components

3.9.1 Dependencies Among Service Instances and Component Service Instances

The Availability Management Framework defines two types of dependencies among 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. Refer to Section 3.2.7 for more details on how SI-SI dependencies pertain to the application logical entity.

The dependencies apply in two cases:

- When a service unit (component, respectively) is assigned the active HA state on behalf of a service instance (component service instance, respectively).
- When the active HA state has been assigned to a service unit (component, respectively) on behalf of a service instance (component service instance, respectively), and another HA state is now assigned, or the active HA state assignment is removed.

3.9.1.1 Dependencies Between 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 3.2.7), 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, etc. are either fully-assigned or partially-assigned (see Section 3.3.3.2).
These dependencies are cluster-wide, which means SI2 and SI3 may or may not belong to the same service group as SI1.

3.9.1.2 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:

'**tolerance time**': In the case of a dependency of a service instance SI1 on the service instance SI2, this time indicates for how long SI1 can tolerate SI2 being in the unassigned state (see Section 3.3.3.2). 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, i.e., it will make SI1 unassigned.

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.

3.9.1.3 Dependencies Between 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. In this case, 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 the configuration.

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.

In other words, 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 any of the components of the service unit in question has already acknowledged the assignment of the active HA state for CSI2 by calling *saAmfResponse()*.

Note that dependencies between component service instances also apply when restarting components within a service unit. Since component service instances assigned to a component must be removed when restarting a component, dependent component service instances within the same service instance will also be removed.

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 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.

3.9.2 Dependencies Between 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 there also exist implicit dependencies between a proxy and its proxied components - not to be discussed here.)

A system administrator can take advantage of such a feature to avoid launching many processes, which perform a lengthy initialization, concurrently, as this could lead to CPU saturation. A "tempered" launching of these processes could be more adequate.

Dependencies between components are configured by associating an **instantiation level** with each pre-instantiable component. The instantiation level is a positive integer configured for such components.

Within a service unit, the Availability Management Framework instantiates the pre-instantiable components according to the instantiation level specified in the Availability Management Framework configuration. 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, 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 here. If single components within a service unit are restarted, the instantiation 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 would assign to them the active HA state for a component service instance).
In case dependencies amongst a non-pre-instantiable and another component exist, they should be resolved by using the inter-CSI (CS I - CSI) dependency scheme.

3.10 Approaches for Integrating Legacy Software or Hardware Entities

There are two ways to integrate non-SA-aware software or hardware entities into the Availability Management Framework model:

- 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 AMF 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:

- (i) 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 usu-

ally requires a very simple redundancy model such as 2N, while the legacy entity may need a more complex redundancy models such as N+M and N-way active.

- (ii) 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 of the cluster, while the proxy entity has to be located in a node.

3.11 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. This includes monitoring the death of processes, which 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()* (refer to Section 6.6.1 on page 196) and *saAmfPmStop()* (refer to Section 6.6.2 on page 198).
External active monitoring is supported with two command line interfaces (CLI) commands, AM_START (refer to Section 4.7 on page 149) and AM_STOP (refer to Section 4.8 on page 150), used to start and 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 6.1.2 on page 159).

3.12 Error Detection, Recovery, Repair, and Escalation Policy

3.12.1 Basic Notions

3.12.1.1 Error Detection

Error detection is the responsibility of all entities in the system. Errors are reported to the Availability Management Framework through the *saAmfComponentErrorReport()* API function. 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. Refer to Section 3.3.2.2 on page 48 for the different cases.

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.

3.12.1.2 Restart

Restarting 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 running the CLEANUP command or by asking the proxy component to do so. Other components are terminated gracefully by first attempting to run the terminate callback or the TERMINATE command.

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 3.3.2.1, which in turn impacts whether its component service instances must be removed. (Refer to Section 3.3.2.3).

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 3.3.2.1) that impacts the service unit's presence state (see Section 3.3.1.1) and consequently readiness state (see Section 3.3.1.4), which determines the service instance assignments. If a service unit contains only restartable components (i.e., *disableRestart*=FALSE), it remains in the in-service readiness state during the restart, thus its service instance assignments remain intact.

3.12.1.3 Recovery

This 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 component's HA state on their behalf.

Recovery actions include:

Different Levels of Restart:
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 may have happened during the recovery, which would prevent the Availability Management Framework from performing such assignments (for example some dependencies may not be satisfied anymore). Two 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 that contains the erroneous component: All components of the service unit are abruptly terminated and then instantiated again (See Section 3.12.1.2). This action is performed as a consequence of an escalation of an SA_AMF_COMPONENT_RESTART recommended recovery action.

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 3.3.2.1 on page 36.

Different Levels of Fail-Over:
Either because the restart recovery action has been disabled in the configuration of a particular component or because previous attempts to restart the component failed,

the Availability Management Framework may decide to recover by reassigning service instances to service units other than the one they are currently assigned to. The different levels of fail-over listed below 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 a node) and how abruptly component service instances are removed from the components they are currently assigned to (regular HA state management leading to the removal of the component service instance, or graceful component termination, or abrupt component termination or abrupt node reboot).

- **Component or Service Unit Fail-Over**
  The Availability Management Framework provides a configuration attribute at the service unit level to indicate if a component fail-over should trigger a fail-over of the entire service unit or only of the erroneous component.
  By default, a service unit fail-over is performed.

  If the service unit is configured to fail over as a single entity, 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, which were assigned to it are failed over. Other components are not terminated but all service instances, which 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 is made of 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 (assigned respectively to C1, C2 and C3) and SI2 contains only 2 CSIs: CSI21 and CSI23 (assigned respectively to C1 and C3).

  Assume that C2 fails. C2 is abruptly terminated. As C2 was assigned CSI12, CSI12 is failed over and the rest of SI1 needs to be switched over: CSI11 and CSI13 are switched over. However, there may be no need to switch over SI2 as it has no CSIs assigned to C2, which failed.

  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, 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 then 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, which were assigned to it. All service instances assigned to service units on the node 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.
  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 failing over 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 through a low level interface without trying to terminate the components individually. The reboot operation must be carried out in a way that puts all local components of the node (including its hardware components) into the uninstantiated presence state. Depending on the physical node configuration, this may require powering-down or resetting some hardware entities (potentially using the HPI). 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.

Note that in case that a component fails, just removing component service instance from it and reassigning them to it (without restarting it) is not considered as a valid recovery action.

One of the recovery methods described in this section is configured per component as a default recovery action (referred to as *recoveryOnError* for convenience) that is engaged under the following circumstances:

- 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.

3.12.1.4 Repair

This is the action which has to be performed on erroneous entities (i.e., with a disabled operational state) to bring them back into a healthy state (i.e., with an enabled operational state). There is a Availability Management Framework configuration attribute at node and service group level that determines if the Availability Management Framework engages in automatic repair or not.

When this configuration attribute is turned on, the Availability Management Framework can perform an automatic repair action after undertaking some recovery actions at the service unit or node levels. If this attribute is turned on at the service group level, it applies to all service units within the service group; if this attribute is turned on at the node level, it applies to only the node.

If the automatic repair configuration attributes are turned off, the Availability Management Framework performs no automatic repair action and it is the responsibility of system management applications or system administrators to perform repair actions which are not under the control of the Availability Management Framework and then reenable the appropriate operational states when the repair is successfully completed using the SA_AMF_ADMIN_REPAIRED administrative operation. It is expected that these repair actions bring the repaired service units in either the instantiated or uninstantiated presence state before reenabling the appropriate operational states.

The following describes, for each recovery action, the automatic repair action which can be performed by the Availability Management Framework.

The Availability Management Framework treats the component and service unit restart recovery actions 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 the service unit when the restart operation completes successfully.

In the case of a component fail-over recovery action, independently of any configuration attribute setting, the Availability Management Framework always tries to reinstantiate the erroneous component and if it is successful, re-enables it. This is performed in order to avoid leaving a service unit partially disabled for an indefinite amount of time.

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 re-joins 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 re-joins the cluster membership if the node has not been rebooted in the meantime.

- Service Unit Failover 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 and all components have been successfully terminated, the Availability Management Framework reenables the operational states of the service unit and 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) and then reinstantiates service units if deemed necessary.
- Node Switch-Over, Fail-Over and Failfast Recovery - After a node switch-over or node fail-over recovery action, if the erroneous node has the automatic repair configuration attribute set, 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) and 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 then reinstantiates service units if deemed necessary.

The following table describes the recovery policies and the associated automatic repair policies.

Table 14 Auto Repair Actions

Recovery Action	Automatic Repair
SU Failover	AMF attempts to instantiate the SU
Node Switch-Over	Node reboot

Table 14 Auto Repair Actions

Node Fail-Over	Node reboot
Node Failfast	None - Already part of recovery

3.12.1.5 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 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).

3.12.2 Recovery Escalation Policy of the Availability Management Framework

3.12.2.1 Recommended Recovery Action

The following recommended recovery actions are supported by the *saAmfComponentErrorReport()* API:

- SA_AMF_NO_RECOMMENDATION: used when the scope of the error is unknown. The configured recovery policy for the component (*recoveryOnError*) is engaged by the Availability Management Framework in such a case.
- SA_AMF_COMPONENT_RESTART: used when the scope of the error is the component.
- 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
  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 return code SA_AIS_ERR_ACCESS, unless the recommended recovery action is SA_AMF_NO_RECOMMENDATION.

The following three levels of escalation are implemented by the Availability Management Framework:

Table 15 Levels of Escalation

Escalation Level	Recommendation	Escalated to
1	SA_AMF_COMPONENT_RESTART	service unit restart
2	SA_AMF_COMPONENT_RESTART	service unit fail-over
3	SA_AMF_COMPONENT_RESTART or SA_AMF_COMPONENT_FAILOVER	SA_AMF_NODE_FAILOVER

3.12.2.2 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 (called the probation period), the Availability Management Framework escalates 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 fails over the entire service unit.

Each service group can be configured with the following parameters:

- *component_restart_probation* (time value)
- *component_restart_max* (maximum count)
- *SU_restart_probation* (time value)
- *SU_restart_max* (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 3.9.2) should not be counted.

If this count does not reach the *component_restart_max* value before the end of the "component restart" probation period (the length of the period is specified by *component_restart_probation*), 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 *component_restart_max* 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 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 *SU_restart_max* value before the end of the "service unit restart" probation period (the length of the period is specified by *SU_restart_probation*), the "service unit restart" probation period for the affected service unit expires.

If this count reaches the *SU_restart_max* 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.

3.12.2.3 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.

- *SU_failover_probation*
- *SU_failover_max*

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, which 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 an error report with an SA_AMF_COMPONENT_FAILOVER recommended recovery action.

If this count does not reach the *SU_failover_max* value before the end of the “service unit fail-over” probation period (the length of the period is specified by *SU_failover_probation),* the “service unit fail-over” probation period is terminated for all service units of the affected node.

If this count reaches the *SU_failover_max* 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 rest of this document, this interface will be referred to as the Component Life Cycle Command Line Interface (CLC-CLI).

Five CLC-CLIs are included in this specification: INSTANTIATE, TERMINATE, CLEANUP, AM_START and AM_STOP.

4.1 Common Characteristics

CLC-CLIs and associated configuration parameters are part of the component configuration as defined for the Availability Management Framework. Here, only the basic semantics associated with such descriptor are described; issues like format, range, and so on, are not explained here.

This descriptor contains for each CLC-CLI at least:

- the path name of the CLC-CLI command,
- the list of environment variables and arguments to be provided to the CLC-CLI by the Availability Management Framework at runtime,
- a timeout value used to control the execution of the CLC-CLI. The Availability Management Framework considers that the CLC-CLI failed if it did not complete in the time interval specified by this timeout.

CLC-CLIs are idempotents.

4.2 CLC-CLI's Environment Variables

- SA-aware components can use regular Availability Management Framework APIs to access the name/value pairs for each component service instance (see Section 6.3.5.4) assigned to itself or to the components it is proxying for. CLC-CLIs of non-proxied, non-SA-aware components can benefit from an easy access to the configuration parameters associated to their assigned component service instances. Therefore, the Availability Management Framework will pass all name/value pairs of the component service instance as environment variables of each CLC-CLIs.
- The SA_AMF_COMPONENT_NAME environment variable is set in the environment of each CLC-CLI. This environment variable contains the name of the component the CLC-CLI 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 ([7]) to obtain a character string.
  - Then, the quoted-printable encoding from RFC 2045 ([8]) is used to substitute non-printable characters in the string.

4.3 Exit Status

The valid range for the exit status is 0 <= exit status <= 255. CLC-CLIs have a zero exit status in case of success, non-zero in case of failure. Values in the range 200 <= exit status <= 254 have either pre-defined 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.

4.4 INSTANTIATE Command

The Availability Management Framework runs the INSTANTIATE command when it wants to instantiate a new instance of a non-proxied, local component.

This command is mandatory for all non-proxied, local components and may not be used for proxied components: A proxied component must be instantiated by its proxy component.

The INSTANTIATE command may create zero, one, or several processes, files, shared memory segments, etc.

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.

INSTANTIATE 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 which 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.

Note that when an SA-aware component unregisters itself with the Availability Management Framework, it does not transition to the uninstantiated presence state.

INSTANTIATE must return a non-zero exit status if the component is not instantiated successfully. If INSTANTIATE returns a non-zero exit status (even if it is outside the range valid for the Availability Management Framework as described in Section 4.3), 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 4.6, to perform all necessary cleanup.

The Availability Management Framework makes few attempts to recover from this error by trying to restart (for instance, 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. If this fails, the Availability Management Framework makes a single attempt to reboot the node to solve the problem. There is a Availability Management Framework configuration parameter at the node level to disable node reboot in this situation.

If node reboot is disabled, or if a single reboot did not solve the problem, the Availability Management Framework sets the component's operational state 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 termination-failed state in this case overrides instantiation-failed state). 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 (beyond the node reboot attempt already performed) is attempted by the Availability Management Framework for this service unit and an explicit administration action is required to repair it.

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.

4.5 TERMINATE Command

SA-aware or proxied components are terminated by the Availability Management Framework by invoking the *saAmfComponentTerminateCallback()* callback function.

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 it could be resumed by another instance of the same component or another component with minimal disruption.

This CLC-CLI is mandatory for all local non-proxied, non-SA-aware components and may not be used for SA-aware components and proxied components.

When the TERMINATE command completes successfully, it must leave the component un-instantiated. The un-instantiated 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.

TERMINATE should release all resources allocated by the component. TERMINATE 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 does not complete in the time period specified by the TERMINATE timeout, the Availability Management Framework runs the CLEANUP command to perform all necessary cleanup actions.

4.6 CLEANUP Command

When recovering from errors, the Availability Management Framework does not trust erroneous components to execute any callbacks, but still needs a method to terminate the particular instance of a component with the minimum interaction with the component itself. The same situation happens when either the *SaAmfComponentTerminateCallbackT* callback (for SA-aware components) or the TERMINATE command (for non-proxied, non-SA-aware components) failed to terminate a component. In this case, the Availability Management Framework forces a cleanup of the component by running the CLEANUP command.

This command is mandatory for all local components (proxied or non-proxied) and may not be used for external components.

When the CLEANUP command completes successfully, it must leave the component un-instantiated. 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 below), the CLEANUP command also needs to cleanup any resources that the AM_STOP command may have failed to cleanup.

If the CLEANUP command returns an error or does not complete in the time period specified by the CLEANUP timeout, the Availability Management Framework has the possibility (controlled by a configuration attribute of the node) to force a node failfast recovery action. The node failfast includes an implicit node reboot that puts all local components of the node (including its hardware components) into the uninstantiated presence state. (see Section 3.12.1.3 for more details)

If the node reboot is not allowed by the node's configuration, the Availability Management Framework sets the component's operational state to disabled and its 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 administration action is required to repair it.

If the component was assigned the active HA state for some CSIs when the CLEANUP command was executed, and semantics of the redundancy model of its enclosing service group guarantees that, at a point in time, only one component can be in the active HA state for a given CSI, the failure to terminate that component prevents the Availability Management Framework to assign another component the active HA state for these CSIs (and by the same token prevents the assignment of other service units active for the service instances that contains the involved CSIs). In this case, the service instances will stay unassigned until an administrative action is performed to terminate the failed component.

4.7 AM_START Command

The Availability Management Framework executes the AM_START command after the component has been successfully instantiated or to resume monitoring after it 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 using the *saAmfComponentErrorReport()* interface.

The AM_START command is optional for all local components and may not be used for external components.

If the AM_START command returns an error or fails to complete in the configured timeout, the Availability Management Framework will retry a few times to start the monitor. If AM_START did not complete in the timeout period, the Availability Management Framework runs AM_STOP before running AM_START again. If after a configurable amount of retries, the Availability Management Framework fails to start the monitor, the Availability Management Framework reports an error on the component level.

4.8 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 requested to do so through administrative operations.

The AM_STOP command is mandatory for components, which have an AM_START command, and may not be used for components, which do not have an AM_START command.

If the AM_STOP command returns an error or fails to complete in the configured timeout period, 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 Availability Management Framework terminates the component and then invokes the CLEANUP command 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.

4.9 Summary of Usage of CLC-CLI Commands Based on the Component Category

Table 16 Usage of CLC-CLI Commands for each Component Category

CLC-CLI command	Mandatory	Forbidden	Optional
INSTANTIATE	non-proxied, local components	proxied components	-
TERMINATE	non-proxied, non-SA-aware components	SA-aware and proxied components	-
CLEANUP	local components	external components	-
AM_START AM_STOP	-	external components	local components

For further details on component categories, refer to Table 2 on page 34.

5 Proxied Component Management

5.1 Assumptions About Proxied/Proxy Components

To make the management of proxied components simpler, the following assumptions are adopted:

- Although the proxied/proxy solution is recommended when the proxied components are located outside of nodes, it is also valid to have proxied components within local service units.
- Pre-instantiable proxied components cannot be located on the same service unit as proxy components. This is devised to prevent potential cyclic dependencies during the service unit instantiations.
- The configuration of proxy/proxied components should include information about the association of a proxied component to the CSI through which the proxied component will be proxied (termed **proxy CSI**). To realize this, a proxied component configuration should have a configuration attribute, which includes 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; some for proxying proxied component sets and others for providing non-proxy services. Note that, functionally, there is no difference between proxy CSIs and other CSIs. The proxy CSI corresponds to the workload of 'proxying' a proxied component and needs to be configured as a configuration attribute of 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.

5.2 Life-Cycle Management of Proxied Components

It is assumed that a proxied component configuration will have information about which CSI is configured to proxy the proxied component. Thus, using this configuration knowledge for all proxied components, the Availability Management Framework determines the associations amongst each proxy CSI and proxied components to be proxied by the CSI.

After the Availability Management Framework successfully assigns a proxy CSI with active HA state to a proxy component, the Availability Management Framework will request the active proxy component to instantiate the corresponding pre-instantiable proxied components using the *SaAmfProxiedComponentInstantiateCallbackT*. It is important to note that 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; hence, this step is not applicable for non-pre-instantiable components.

After a proxied component is instantiated, the respective proxy component should register the proxied component with the Availability Management Framework. 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 the proxied component, the Availability Management Framework shall assign CSI(s) to those registered proxied components (via the respective proxy component), with appropriate HA states. If it so happens that certain proxied components fail to register after the instantiation phase, then the CSI(s) for such components are not assigned by the Availability Management Framework. Subsequently, the Availability Management Framework should try to revive the failed component by invoking the *SaAmfProxiedComponentCleanupCallbackT* API and reinstantiating it in the same way as it would for an SA-aware component (see Section 4.4). Also refer to Appendix C, which uses a sample configuration to illustrate a typical proxy and 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 (Refer Section 6.5.1 for component registration interface).

5.3 Proxy Component Failure Handling

If a proxy component fails, the Availability Management Framework may perform a fail-over, if it 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 proxied component failure. The implicit unregistration simply indicates that the proxy component is unable to continue 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, which is capable of proxying the given proxied components, then it will make an active assignment of the proxy CSI(s) of the proxied components to this other proxy component. The proxy component, which is chosen to take over the proxy job, is selected based on the redundancy model of the service group containing the proxy compo-

nent. For example, for a proxy component contained in a service unit, which 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 which had standby HA assignment for the proxy CSI (i.e., 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 the newly selected proxy component re-registers the proxied component without an explicit instantiation step.

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 and an appropriate alarm shall be issued by the Availability Management Framework (Refer Section 8) to indicate this situation. Whenever a proxied component enters the *SA_AMF_PROXY_STATUS_PROXIED* status, an appropriate notification will be issued to indicate the change in the status of the proxied component.

6 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 on page 23. 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
- Availability Management (Component Service Instance Management)
- Component Life Cycle
- Protection Group Management
- Error Reporting
- Component Response to 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 (via 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 in specifying the configuration to describe the relationships between the components that the Availability Management

Framework has to maintain. Configuration and administrative APIs are the subject of future specifications.

6.1 Availability Management Framework Model for the APIs

6.1.1 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()* function, defined in Section 6.4.1 on page 184. 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()* function is the set of callback functions associated with this initialization.

One of the processes of SA-aware component registers the component it represents with the Availability Management Framework, using the *saAmfComponentRegister()* function, defined in Section 6.5.1 on page 189, providing the handle returned by the *saAmfInitialize()* function.
A component is registered with the Availability Management Framework to inform the Availability Management Framework that the component is ready to provide service, i.e., to provide services for component service instances. Conversely, a component may unregister with the Availability Management Framework only when either

- the unregistration is done for the component to perform diagnosis and repair on its own in case it is no longer able to provide service, possibly due to a fault, or
- it is explicitly instructed by the Availability Management Framework to be terminated or restarted (See Section 7.4.7 on page 242). In this case, the Availability Management Framework does not disable the component.

What has been said in this section is only applicable for non-proxied components because if a proxy component unregisters one of its proxied components, the component service instances for the proxied components are not removed but rather kept assigned. Moreover, the operational and HA states of the proxied components do not change. This is motivated by the assumption that another proxy will soon take over the role of the previous proxy.

A proxy component must first register itself and then register one or more proxied components on their behalf.

A process, that is part of a proxy component and that registers several proxied components, may issue several calls to the *saAmfInitialize()* function to provide different sets of callback functions and obtain different handles that can be used to register the various proxied components.

The process of an SA-aware component COMP that registers a component (SA-aware or proxied) is called the **registered** process for this component and the other processes of the component COMP are named **unregistered** processes. There is a set of callback functions that are called by the Availability Management Framework in the context of the registered process only. Additionally, there are other API functions that may be called only by a registered process. If an API function may only be called by a registered process or if a callback function may only be invoked for the registered process, this is made explicit in the description of the APIs of the Availability Management Framework. Appendix B provides a table showing which functions may be invoked by unregistered 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()* function, defined in Section 6.12.1 on page 226, 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()* function. The *saAmfResponse()* function must identify the callback action with which it is associated by providing the *invocation* parameter that the Availability Management Framework supplied in the callback.
- Any function of the Availability Management Framework API, including *saAmfResponse(),* can be invoked from callback functions.

6.1.2 Component Healthcheck Monitoring

6.1.2.1 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 (key) that is associated with a set of configuration attributes. Though some of the health-

check attributes can be specified at the component level, the deployer may finalize healthcheck attributes for each component instance. Healthchecks can be invoked by the Availability Management Framework or by the component.

The issue of potential transient overload caused by (or affected) 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.

6.1.2.2 Healthcheck Types

There are two types of healthcheck depending on the invoker of the healthcheck:

- Framework-invoked healthcheck: With this type of healthcheck, the Availability Management Framework invokes the *saAmfHealthcheckCallback()* callback periodically according to the healthcheck configuration attributes. The Availability Management Framework expects the component to respond to an invoked healthcheck by replying to the healthcheck invocation through *SaAmfResponse()*.
- Component-invoked healthcheck: This type of healthcheck is invoked by the component itself (according to its configured parameters), and the result of the healthcheck is reported to the Availability Management Framework through *saAmfHealthcheckConfirm()*.

6.1.2.3 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 which 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, i.e., it is possible to start healthchecks before the component is registered or after the component is unregistered.

6.1.2.4 Healthcheck Configuration Issues

Only the basic semantics of the configuration attributes of healthchecks are treated here; issues like syntax, range, and so on, are left to the upcoming SA Forum System Management Specification. The Availability Management Framework retrieves the healthcheck configuration via the *healthcheckKey* parameter, specified in the healthcheck API calls. The scope of the *healthcheckKey* is limited to the component and is not cluster-wide.

It is assumed that the component configuration retrieved by the Availability Management Framework has gone through a series of sanity checks and configuration validations before the cluster startup. Hence, this rules out errors like specifying too frequent healthcheck in the configuration. Also based on these validations, the only reason that the Availability Management Framework may reject a healthcheck start request is when some of given parameters such as component name or *healthcheckKey* are invalid.

A healthcheck configuration comprises two attributes:

- *period*: This indicates the period at which the corresponding healthcheck should be initiated. This attribute is defined for both the framework-invoked and the component-invoked healthchecks; however, it has different meanings for these two types of healthchecks as will be explained below.
- *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. This is applied only for the framework-invoked healthcheck type.

The component developer is aware of the healthcheck type supported by a component, and the component developer specifies this healthcheck type in the corresponding healthcheck API calls.
The *period* and *maximum-duration* configuration attributes are specified at deployment time.

Role of *period* and *maximum-duration* in the 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, for each healthcheck, there is **at most** one callback invocation pending for the response, at any given time. Of course, as a process may have

started several healthchecks in parallel, the Availability Management Framework will invoke callbacks for these different healthchecks, independently. In the next bullet, it is described what happens when a process does not respond timely to a framework-invoked healthcheck.

- *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 guarantees that in normal conditions with expected load, the response of the 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 the period is set too short, the Availability Management Framework may consider the component of a healthy process, running in highly load environment, as faulty. On the other hand, if *period* is set too large, then 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 process's response for 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,

then 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, then the latency for the detection of a faulty component being healthchecked may be increased.

Role of period in the component-invoked healthchecks

As already explained, component-invoked healthchecks do not have the *maximum-duration* attribute (if it is given, 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 healthchecks periodically via *saAmfHealthcheckConfirm()* calls, no later than at the end of every period. More specifically, the Availability Management Framework reports an error on the component if it does not receive a healthcheck confirmation from the component before the end of every period. The recommended recovery for this error was specified by the process when it invoked the *saAmfHealthcheckStart()* call. The deployer should add enough slack time to period such that the healthcheck invoked by a healthy process can reach the Availability Management Framework on-time.

6.1.3 Availability Management (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 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 are limited to the management of the HA 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.
- Change the HA state of a component on behalf of a component service instance (active, standby, quiescing, quiesced).

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 first request is acknowledged by the component (this is done by using the *saAmfResponse()* API function, as described in Section 6.11.1). 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 state. For details, refer to the readiness state in Section 3.3.2.3 and to the HA state in Section 3.3.2.4.

The component service instance management comprises data structures and APIs. The API functions are described in Section 6.8 on page 206.

6.1.4 Component Life Cycle Management

In this section, the callback function to request a component to terminate is described. This section contains also additional callback functions that proxy components export to enable the Availability Management Framework to manage proxied components. The API functions are described in Section 6.9 on page 212.

6.1.5 Protection Group Management

The basic concepts have been explained in Chapter 3. For the API functions, refer to Section 6.10 on page 216.

6.1.6 Error Reporting

For the API interfaces, refer to Section 6.11 on page 223.

6.1.7 Component Response to Framework Requests

For the API interfaces, refer to Section 6.12 on page 226.

6.1.8 API Usage Illustrations

This section illustrates the usage of the Availability Management Framework API by different categories of components.

Figure 23 next shows an example of a local component consisting of a single process. The numbers in circles indicate the sequence of events in time.

Figure 23 Local SA-Aware Component Consisting of a Single Process

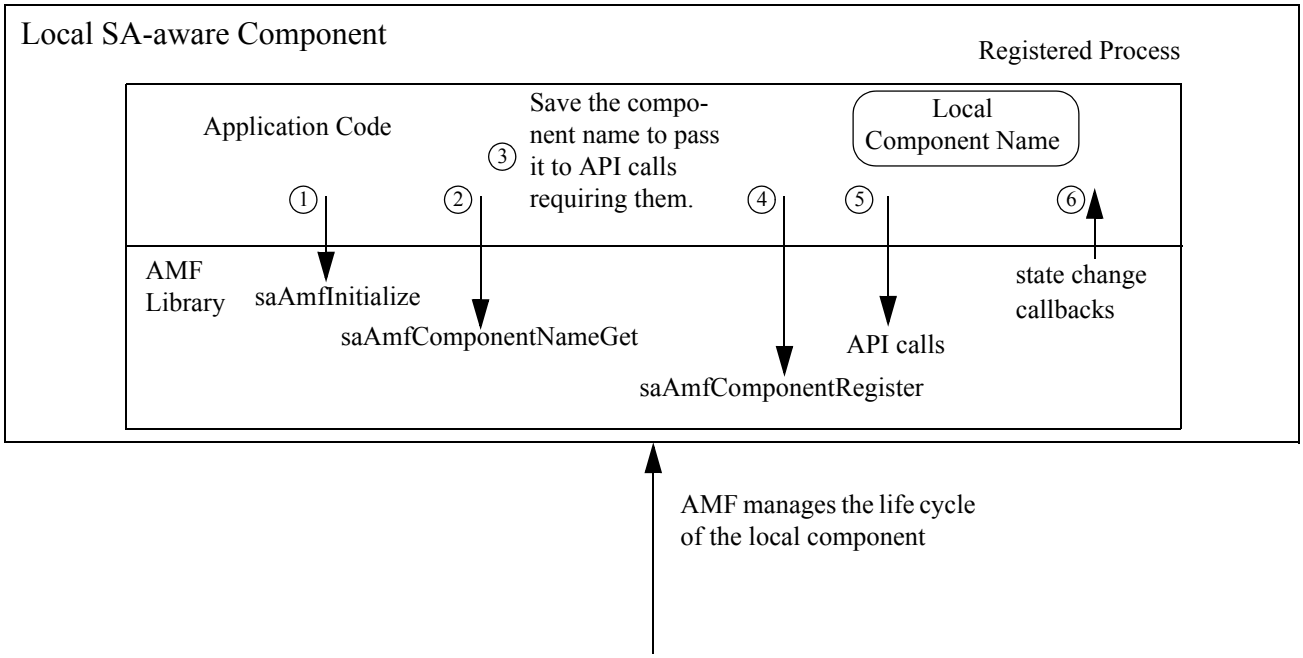


Figure 24 next shows an example of an SA-aware component consisting of multiple processes. The numbers in circles indicate the sequence of events in time.

Figure 24 Local SA-Aware Component Consisting of Multiple Processes

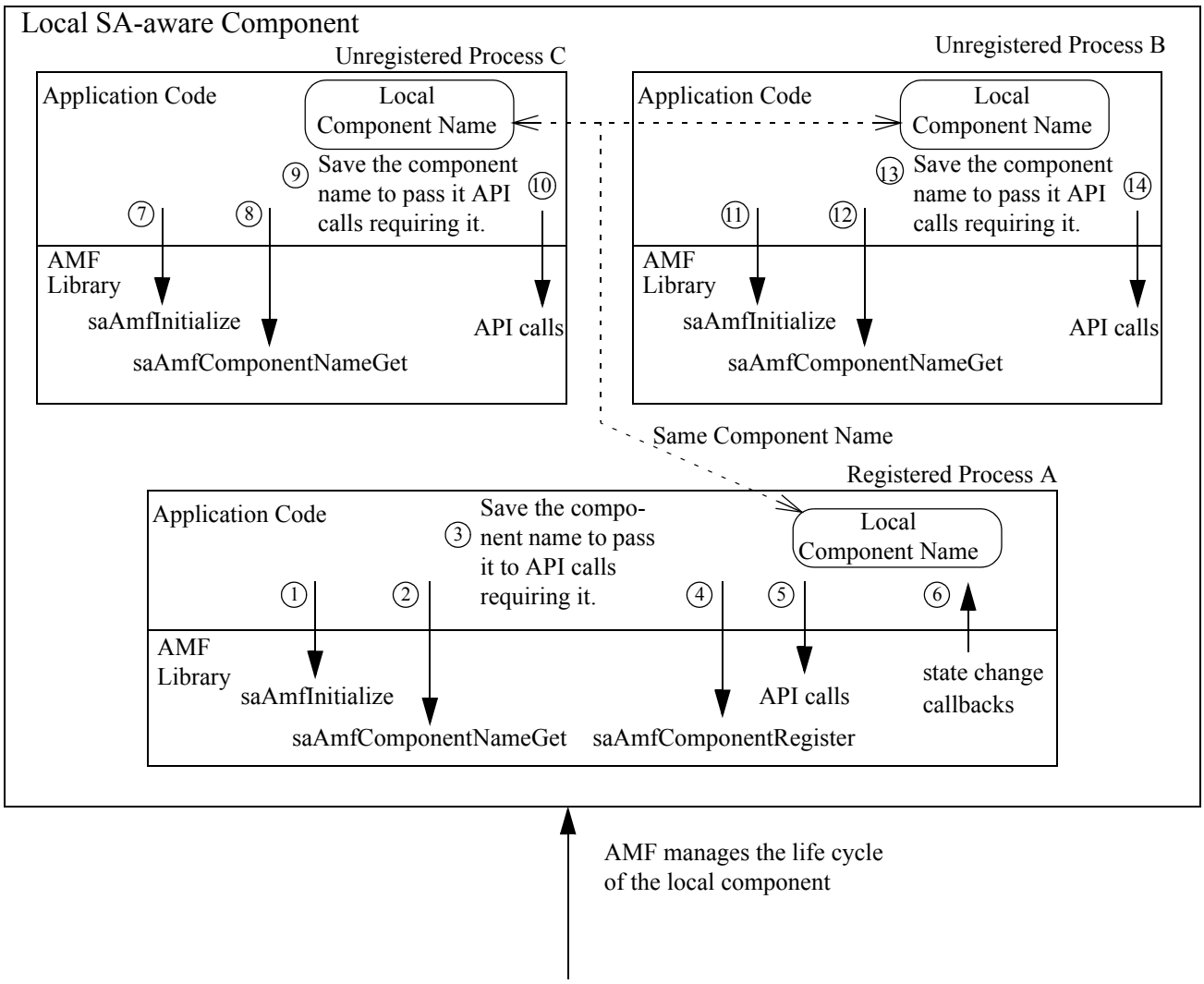
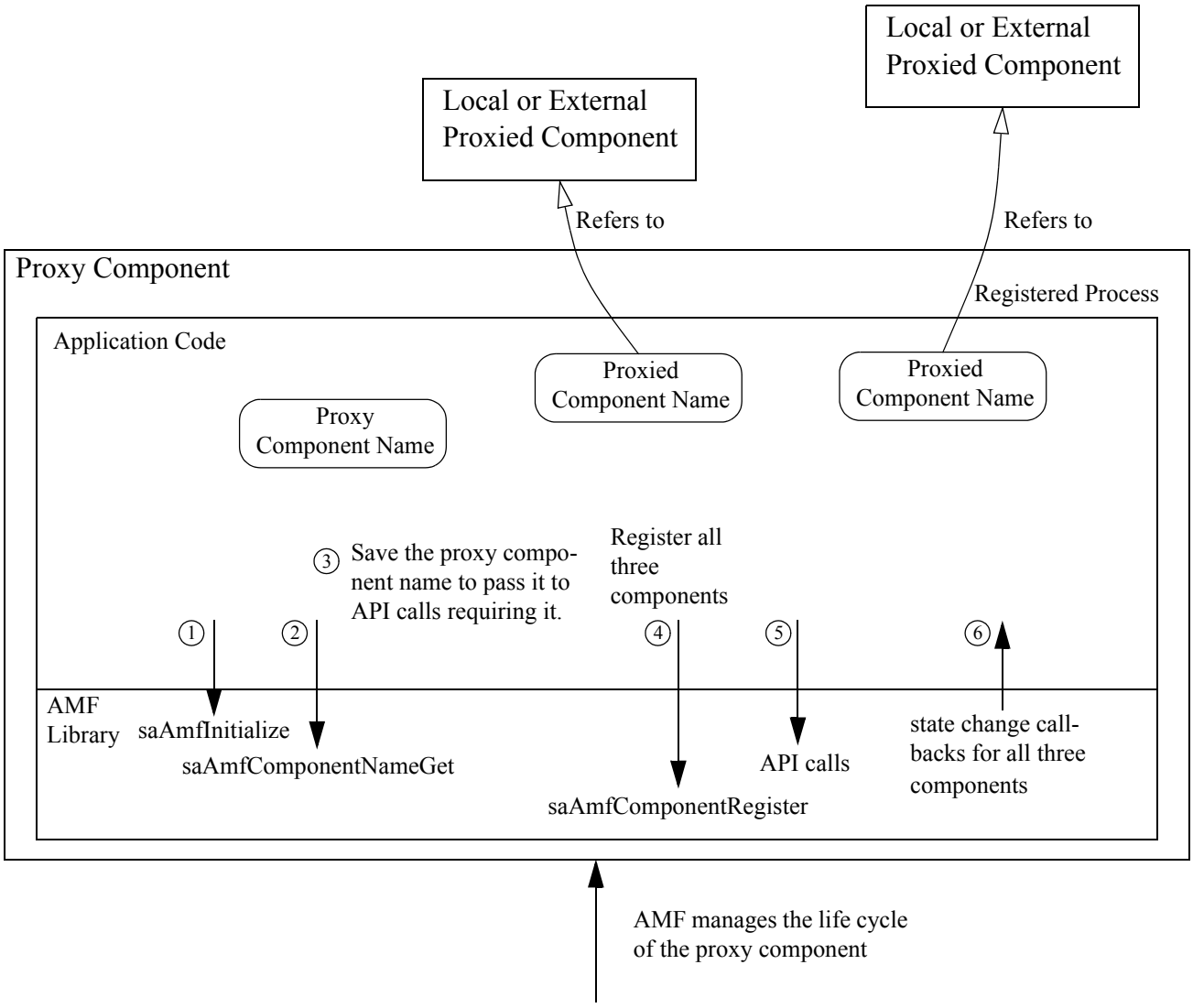


Figure 25 next shows an example of a single-process proxy component that registers itself and two proxied components with the Availability Management Framework. The numbers in circles indicate the sequence of events in time.

Figure 25 **A Single-Process Proxy Component and Two Proxied Components**



6.2 Include File and Library Names

The following statement 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>

(for saAmfComponentErrorReport() and saAmfComponentErrorClear())

To use the Availability Management Framework API, an application must be bound with the following library:

libSaAmf.so

6.3 Type Definitions

The Availability Management Framework uses the types described in the following sections.

6.3.1 SaAmfHandleT

typedef SaUint64T SaAmfHandleT;

The handle to the Availability Management Framework that a process acquires through the *saAmfInitialize()* function and uses in subsequent invocations of the functions of the Availability Management Framework.

6.3.2 Component Process Monitoring

This section describes the data types required for the passive monitoring of processes of a component by the Availability Management Framework.

6.3.2.1 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;

6.3.2.2 SaAmfPmStopT type

```
typedef enum {
    SA_AMF_PM_PROC = 1,
    SA_AMF_PM_PROC_AND_DESCENDENTS = 2,
    SA_AMF_PM_ALL_PROCESSES = 3
} SaAmfPmStopQualifierT;
```

Refer to Section 6.6.2 on page 198 for explanation of the enum values in *SaAmfPmStopQualifierT.*

6.3.3 Component Healthcheck Monitoring

6.3.3.1 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.

6.3.3.2 SaAmfHealthcheckKeyT

#define SA_AMF_HEALTHCHECK_KEY_MAX 32

typedef struct {

SaUint8T key[SA_AMF_HEALTHCHECK_KEY_MAX];

SaUint16T keyLen;

} SaAmfHealthcheckKeyT;

6.3.4 Types for State Management

6.3.4.1 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.

6.3.4.2 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.

6.3.4.3 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.

6.3.4.4 Operational State

```
typedef enum {
    SA_AMF_OPERATIONAL_ENABLED = 1,
    SA_AMF_OPERATIONAL_DISABLED = 2
}SaAmfOperationalStateT;
```

The operational state is enabled or disabled.

6.3.4.5 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.

6.3.4.6 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 a SI is unassigned, fully-assigned or partially-assigned.

6.3.4.7 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 or unproxied.

6.3.4.8 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

} SaAmfStateT;

All defined states are readiness, HA state, presence, operational, and administrative.

6.3.5 Component Service Instance Types

6.3.5.1 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. For assignments, this flag is set for components providing the 'x_active_or_y_standby' capability model. The Availability Management Framework can use this flag in other cases 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*.

6.3.5.2 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 assignment is not the result of a switch-over or fail-over of the specified component service instance from another component to this component. At least one other component is still active for this component service instance. This flag is used, for example, in the N-way active redundancy model when a new component is assigned active for a component service instance while other components are already assigned active for that component service instance.

6.3.5.3 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 information shown below:

- The Availability Management Framework uses the *transitionDescriptor* that is appropriate for the redundancy model of the service group this component belongs to.
- 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 which is still assigned the active HA state for all targeted component service instances. The choice of the component selected in that case is arbitrary.

typedef struct {

SaNameT activeCompName;

SaUint32T standbyRank;

} SaAmfCSIStandbyDescriptorT;

The fields of the *SaAmfCSIStandbyDescriptorT* structure have the following interpretation:

- *activeCompName* - This is the name of the component that is currently active for the one or all of the specified component service instances. The name is empty if no active component exists.
- *standbyRank* - The 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 where several components may assume the standby HA state for the same component service instance at the same time, *standbyRank* indicates to the component which 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.

6.3.5.4 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 for a single component service instance. The *attr* pointer points to an array of *number* elements of *SaAmfCSIAttributeT* attribute descriptors.

6.3.5.5 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.

6.3.6 Types for Protection Group Management

6.3.6.1 SaAmfProtectionGroupMemberT

typedef struct {

SaNameT compName;

SaAmfHAStateT haState;

SaUint32T rank;

} SaAmfProtectionGroupMemberT;

The fields of the *SaAmfProtectionGroupMemberT* structure have the following interpretation:

- *compName* - The name of the component that is a member of the protection group.
- *haState* - The *haState* of the member component for the component service instance supported by the member component.
- *rank*- The rank of the member component in the protection group if *haState* is standby.

6.3.6.2 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()* function, defined in Section 6.10.1, is either
  - SA_TRACK_CURRENT or
  - SA_TRACK_CHANGES and the member component was already a member of the protection group in the previous *saAmfProtectionGroupTrackCallback()* callback call, and the component service instance has not been removed from the member component,

and neither *haState* nor *rank* of the *saAmfProtectionGroupMemberT* 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* or *rank* of the *SaAmfProtectionGroupMemberT* structure for the member component have changed.

6.3.6.3 SaAmfProtectionGroupNotificationT

```
typedef struct {
    SaAmfProtectionGroupMemberT member;
    SaAmfProtectionGroupChangesT change;
} SaAmfProtectionGroupNotificationT;
```

The fields of the *SaAmfProtectionGroupNotificationT* 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

6.3.6.4 SaAmfProtectionGroupNotificationBufferT

```
typedef struct {
    SaUint32T numberOfItems;
    SaAmfProtectionGroupNotificationT *notification;
} SaAmfProtectionGroupNotificationBufferT;
```

The fields of the *SaAmfProtectionGroupNotificationBufferT* structure have the following interpretation:

- *numberOfItems* - number of elements of type *SaAmfProtectionGroupNotificationT* in the notification buffer
- *notification* - start address of the notification buffer

6.3.7 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

} SaAmfRecommendedRecoveryT;

The values of this enumeration type have the following interpretation:

- SA_AMF_NO_RECOMMENDATION - This report makes no recommendation for recovery. However, the Availability Management Framework should engage the configured per-component recovery policy (*recoveryOnError*) 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 on the node are failed over to other nodes (via 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 via 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 it is not required to reassign service instances (refer additionally to Section 7.4.7 on page 242). This recommendation should be used when the failure is deemed to be a global application failure. 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.
- 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 through a low level interface without trying to terminate the components individually. To be effective, this operation must be performed such that all nodes are first halted before any of the nodes boots 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.

6.3.8 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
typedef SaUint32T saAmfCompCategoryT;
```

6.3.9 saAmfRedandancyModelT

```
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_REDUNDACY_MODEL = 4,
        SA_AMF_NO_REDUNDANCY_MODEL= 5
} saAmfRedandancyModelT;
```

Refer to Section 3.7 on page 69 for a description of the various redundancy models described in the above enum.

6.3.10 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;
```

Refer to Section 3.6 on page 68 for a description of the values described in the above enum.

6.3.11 Notification Related Types

```
typedef enum {
        SA_AMF_NODE_NAME = 1,
        SA_AMF_SI_NAME = 2
}SaAmfAdditionalInfoIdT;
```

The preceding types are used in Availability Management Framework Alarms and Notifications to convey additional information elements in the “Additional Information ID” field associated with alarms and notifications.

6.3.12 SaAmfCallbacksT

```
typedef struct {
        SaAmfHealthcheckCallbackT
                saAmfHealthcheckCallback;
        SaAmfComponentTerminateCallbackT
                saAmfComponentTerminateCallback;
        SaAmfCSISetCallbackT
                saAmfCSISetCallback;
        SaAmfCSIRemoveCallbackT
                saAmfCSIRemoveCallback;
        SaAmfProtectionGroupTrackCallbackT
                saAmfProtectionGroupTrackCallback;
        SaAmfProxiedComponentInstantiateCallbackT
                saAmfProxiedComponentInstantiateCallback;
        SaAmfProxiedComponentCleanupCallbackT
                saAmfProxiedComponentCleanupCallback;
} SaAmfCallbacksT;
```

The *SaAmfCallbacksT* structure defines the various callback functions that the Availability Management Framework may invoke on a component.

6.4 Library Life Cycle

6.4.1 saAmfInitialize()

Prototype

SaAisErrorT saAmfInitialize(

*SaAmfHandleT *amfHandle,*

*const SaAmfCallbacksT *amfCallbacks,*

*SaVersionT *version*

);

Parameters

amfHandle - [out] A pointer to the handle designating this particular initialization of the Availability Management Framework that is to be returned by the Availability Management Framework.

amfCallbacks - [in] If *amfCallbacks* is set to NULL, no callbacks are registered; otherwise, it is a pointer to an *SaAmfCallbacksT* structure, containing the callback functions of the process that the Availability Management Framework may invoke. Only non-NULL callback functions in this structure will be registered.

version - [in/out] As an input parameter, *version* is a pointer to the required Availability Management Framework version. In this case, *minorVersion* is ignored and should be set to 0x00.
As an output parameter, the version actually supported by the Availability Management Framework is delivered.

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 *amfHandle* is returned 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.
The *amfCallbacks* parameter designates the callbacks that the Availability Management Framework can invoke.

If the implementation supports the required *releaseCode,* and a major version >= the required *majorVersion*, SA_AIS_OK is returned. In this case, 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 above mentioned condition cannot be met, SA_AIS_ERR_VERSION is returned, and the *version* parameter is set to:

if (implementation supports the required *releaseCode*)

releaseCode = required *releaseCode*

else *{*

if (implementation supports *releaseCode* higher than the required *releaseCode*)

releaseCode = the least 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 less 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* parameter is not compatible with the version of the Availability Management Framework implementation.

See Also

saAmfFinalize()

6.4.2 saAmfSelectionObjectGet()

Prototype

```
SaAisErrorT saAmfSelectionObjectGet(
        SaAmfHandleT amfHandle,
        SaSelectionObjectT *selectionObject
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

selectionObject - [out] A pointer to the operating system handle that the process can use to detect pending callbacks.

Description

This function returns the operating system handle *selectionObject,* associated with the handle *amfHandle*. The process can use this operating system handle to detect pending callbacks, instead of repeatedly invoking *saAmfDispatch()* 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 *selectionObject* 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).

See Also

saAmfInitialize(), saAmfDispatch()

6.4.3 saAmfDispatch()

Prototype

```
SaAisErrorT saAmfDispatch(
        SaAmfHandleT amfHandle,
        SaDispatchFlagsT dispatchFlags
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

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, as defined in the SA Forum Overview document.

Description

This function invokes, in the context of the calling thread, 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.

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.

See Also

saAmfInitialize(), saAmfSelectionObjectGet()

6.4.4 saAmfFinalize()

Prototype

```
SaAisErrorT saAmfFinalize(
        SaAmfHandleT amfHandle
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

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()* before it invokes this function. A process must call this function once for each handle it acquired by invoking *saAmfInitialize()*.

If the *saAmfFinalize()* function returns successfully, the *saAmfFinalize()* function releases all resources acquired when *saAmfInitialize()* 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()* is called, the selection object is 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()

6.5 Component Registration and Unregistration

The following functions are used to register and unregister components with the Availability Management Framework.

6.5.1 saAmfComponentRegister()

Prototype

```
SaAisErrorT saAmfComponentRegister(
        SaAmfHandleT amfHandle,
        const SaNameT *compName,
        const SaNameT *proxyCompName
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework. The Availability Management Framework must maintain the list of components registered via each such handle.

compName - [in] A pointer to the name of the component to be registered.

proxyCompName - [in] A pointer to the name of the proxy component that is registering the proxied component, identified by *compName*. The *proxyCompName* parameter is used only when a proxied component is being registered by a proxy component; otherwise, it must be set to NULL.

Description

The *saAmfComponentRegister()* function can be used by an SA-aware component to register itself with the Availability Management Framework. It can also be used by a proxy component to register a proxied component.

An SA-aware component calls *saAmfComponentRegister()* in order to indicate to the Availability Management Framework that it is ready to take component service instance assignments.

The process of an SA-aware component that registers a (possibly different) component is called the **registered** process for the registered component. The other processes of the SA-aware component are called **unregistered** processes.

A registered process for an SA-aware or proxied component differs from the unregistered processes in that some of the API functions and callbacks are only valid for the registered process. Refer to Appendix B for a detailed listing of the various APIs and callbacks that are valid for unregistered processes.

The registered process must have supplied, in its *saAmfInitialize()* call, the *saAmfCSISetCallback()*, *saAmfCSIRemoveCallback()* and *saAmfComponentTerminateCallback()* callback functions.

The registered process for a proxied component must have also supplied, in its *saAmfInitialize()* call, the *saAmfProxiedComponentInstantiateCallback()* and *saAmfProxiedComponentCleanupCallback()* callback functions.

A component (SA-aware or proxied) cannot register or (be registered) twice before having (been) unregistered, even with a different handle, obtained via the *saAmfInitialize()* call.

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 it itself does not fail. In case the proxied component fails, it is the task of the proxy to explicitly unregister the failed component, if this is desired.

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 initialization with *saAmfInitialize()* was incomplete, since one or more of the callback functions that are listed below were not supplied:

- If a component registers itself: *saAmfComponentTerminateCallback(), saAmfCSISetCallback(),* and *saAmfCSIRemoveCallback()*
- If a proxy component registers another component: *saAmfProxiedComponentInstantiateCallback()* and *saAmfProxiedComponentCleanupCallback()*

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly. In particular, this value returned if either *compName* is not a configured component, or *compName* or *proxyCompName* are not DNs or the types of their first RDNs are not *safComp*.

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 *proxyCompName,* has not been registered previously.

SA_AIS_ERR_EXIST - The component, identified by *compName,* has been registered previously, either via the *amfHandle* handle or another handle, obtained via the *saAmfInitialize()* call.

SA_AIS_ERR_BAD_OPERATION - The proxy component, identified by *proxyCompName*, 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.

See Also

saAmfComponentUnregister(), SaAmfCSISetCallbackT, SaAmfCSIRemoveCallbackT, SaAmfComponentTerminateCallbackT, SaAmfProxiedComponentInstantiateCallbackT, SaAmfProxiedComponentCleanupCallbackT, saAmfInitialize()

6.5.2 saAmfComponentUnregister()

Prototype

```
SaAisErrorT saAmfComponentUnregister(
        SaAmfHandleT amfHandle,
        const SaNameT *compName,
        const SaNameT *proxyCompName
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component to be unregistered.

proxyCompName - [in] A pointer to the name of the proxy component unregistering the proxied component identified by *compName*. This parameter is used only for unregistering a proxied component by a proxy component; otherwise, it must be set to NULL.

Description

The *saAmfComponentUnregister()* function can be used for two purposes:

A proxy component can unregister one of its proxied components or an SA-aware component can unregister itself.

The former case will usually apply to enable another proxy to register for the proxied component. Recall that at a given time at most one proxy can exist for a component.

The latter case is used by an SA-aware component to indicate to the Availability Management Framework that it is unable to continue providing the service, possibly because of a fault condition that is hindering its ability to provide service.

When an SA-aware component unregisters with the Availability Management Framework, the framework treats such an unregistration as an error condition (similar to one signaled by an *saAmfComponentErrorReport()*) and engages the configured default recovery action (*recoveryOnError*) on the component. As a consequence, its operational state may become disabled (refer to Section 6.1.1), and, therefore, all of its component service instances are removed from it.

If a proxy component unregisters one of its proxied components, the operational state of the latter does not change because unregistration does not indicate a failure in this case. This is motivated by the assumption that another proxy will soon take over the role of the previous proxy.

During its life cycle, an SA-aware component can register or unregister multiple times. Also a proxy component can register or unregister a proxied component multiple times.

Before unregistering itself, a proxy component must unregister all of its proxied components.

It is understood that a failed component is implicitly unregistered while it is cleaned up.

The *amfHandle* in the *saAmfComponentUnregister()* call must be the same as that used in the corresponding *saAmfComponentRegister()* call.

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, has already been finalized, or the component has not been registered using this handle.

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 proxy component, identified by *proxyCompName,* or the proxied component, identified by *compName,* has not been registered previously.

SA_AIS_ERR_BAD_OPERATION - The requested unregistration is not acceptable because:

- The component identified by *proxyCompName* is not the proxy of the proxied component identified by *compName,* or
- The component identified by *compName* has not unregistered its proxied components before unregistering itself.

See Also

saAmfComponentRegister(), saAmfInitialize()

6.5.3 saAmfComponentNameGet()

Prototype

```
SaAisErrorT saAmfComponentNameGet(
        SaAmfHandleT amfHandle,
        SaNameT *compName
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [out] A pointer to the name of the component to which the process invoking this function belongs.

Description

This function returns the name of the component the calling process belongs to. This function can be invoked by the process before its component has been registered with the Availability Management Framework by *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 which 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 invokes the INSTANTIATE command to create a component. It is the responsibility of the INSTANTIATE commands, and more generally of any entity, which 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 4.2 on page 145.

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.

See Also

saAmfComponentRegister(), saAmfInitialize()

6.6 Passive Monitoring of Processes of a Component

This section describes the API functions, which enable components to request passive monitoring of their processes by the Availability Management Framework.

6.6.1 saAmfPmStart()

Prototype

SaAisErrorT saAmfPmStart(

SaAmfHandleT amfHandle,

*const SaNameT *compName,*

SaUint64T processId,

SaInt32T descendentsTreeDepth,

SaAmfPmErrorsT pmErrors,

SaAmfRecommendedRecoveryT recommendedRecovery

);

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component to which the monitored processes belong.

processId - [in] Identifier of a process to be monitored.

descendentsTreeDepth - [in] Depth of the tree of descendents of the process, designated by *processId,* to be also monitored.

- 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.

- SA_AMF_PM_NON_ZERO_EXIT requests the monitoring of processes exiting with a non-zero 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 6.3.7 on page 180.

Description

The *saAmfPmStart*() function requests the Availability Management Framework to start passive monitoring of specific errors, which may occur to a process and its descendents. Currently, only death of processes can be monitored. If one of the errors being monitored occurs for the process or one of its descendents, the Availability Management Framework will automatically report an error on the component identified by *compName* (see s*aAmfComponentErrorReport()* for details regarding error reports). 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 *compName,* 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.

See Also

saAmfPmStop(), saAmfComponentErrorReport(), saAmfInitialize()

6.6.2 saAmfPmStop()

Prototype

```
SaAisErrorT saAmfPmStop(
    SaAmfHandleT amfHandle,
    const SaNameT *compName,
    SaAmfPmStopQualifierT stopQualifier,
    SaInt64T processId,
    SaAmfPmErrorsT pmErrors
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component to which the monitored processes belong.

stopQualifier - [in] Qualifies which processes should stop being monitored.

- 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 stop monitoring of all processes, which belong to the component identified by *compName*.

processId - [in] Identifier of the process for which passive monitoring is to be stopped.

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.

Description

The *saAmfPmStop()* function requests the Availability Management Framework to stop passive monitoring of specific errors, which 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 *compName,* 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*.

See Also

saAmfInitialize(), saAmfPmStart()

6.7 Component Health Monitoring

The following calls are used to monitor the health of a component.

6.7.1 saAmfHealthcheckStart()

Prototype

```
SaAisErrorT saAmfHealthcheckStart(
    SaAmfHandleT amfHandle,
    const SaNameT *compName,
    const SaAmfHealthcheckKeyT *healthcheckKey,
    SaAmfHealthcheckInvocationT invocationType,
    SaAmfRecommendedRecoveryT recommendedRecovery
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component to be healthchecked.

healthcheckKey - [in] The key of the healthcheck to be executed. Using this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters.

invocationType - [in] This parameter indicates whether the Availability Management Framework or the process itself will invoke the healthcheck calls.

recommendedRecovery - [in] Recommended recovery to be performed by the Availability Management Framework if the component fails a healthcheck. For details, refer to Section 6.3.7 on page 180.

Description

This function starts healthchecks via the invoking process for the component designated by *compName.* The type 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()* call.
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.

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 initialization with *saAmfInitialize()* was incomplete, since 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 - Either one or both of the cases that follow apply:

- The Availability Management Framework is not aware of a component designated by *compName*.
- The healthcheck with *healthcheckKey* is not configured for the component named *compName*.

SA_AIS_ERR_ACCESS - The Availability Management rejects the requested recommended recovery.

SA_AIS_ERR_EXIST - The healthcheck has already been started on the handle *amfHandle* for the component, designated by *compName,* and the same value of *healthcheckKey*.

See Also

SaAmfHealthcheckCallbackT, saAmfHealthcheckConfirm(), saAmfHealthcheckStop(), saAmfInitialize()

6.7.2 SaAmfHealthcheckCallbackT

Prototype

```
typedef void (*SaAmfHealthcheckCallbackT)(
        SaInvocationT invocation,
        const SaNameT *compName,
        SaAmfHealthcheckKeyT *healthcheckKey
);
```

Parameters

invocation - [in] The particular invocation of this callback function. The invoked process returns i*nvocation* when it responds to the Availability Management Framework by invoking the *saAmfResponse()* function.

compName - [in] A pointer to the name of the component that must undergo the particular healthcheck.

healthcheckKey - [in] The key of the healthcheck to be executed.

Description

The Availability Management Framework requests the component, identified by *compName,* to perform a healthcheck specified by *healthcheckKey*. The Availability Management Framework may ask a proxy component to execute a healthcheck on one of its proxied components.
This callback is invoked in the context of a thread issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when starting the healthcheck operation via the *saAmfHealthcheckStart()* call. The Availability Management Framework provides the *invocation* parameter that the invoked process uses in the *saAmfResponse()* function to notify the Availability Management Framework that it has completed the healthcheck. The component reports the result of its healthcheck to the Availability Management Framework using the *error* parameter of the *saAmfResponse()* function, which in this case has 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 using *saAmfComponentErrorReport()*.

If the invoked process does not respond with the *saAmfResponse()* function within a configured time interval or returns an error, the Availability Management Framework must engage the configured recovery policy (*recoveryOnError*) for the component to which the process belongs.

See Also

saAmfResponse(), saAmfHealthcheckStart(), saAmfComponentErrorReport(), saAmfDispatch()

6.7.3 saAmfHealthcheckConfirm()

Prototype

```
SaAisErrorT saAmfHealthcheckConfirm(
    SaAmfHandleT amfHandle,
    const SaNameT *compName,
    const SaAmfHealthcheckKeyT *healthcheckKey,
    SaAisErrorT healthcheckResult
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component the healthcheck result is being reported for.

healthcheckKey - [in] The key of the healthcheck whose result is being reported. Using this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters.

healthcheckResult - [in] This parameter indicates the result of the healthcheck performed by the component. It 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 using *saAmfComponentErrorReport()*.

Description

This function allows a process to inform the Availability Management Framework that it has performed the healthcheck identified by *healthcheckKey* for the component designated by *compName,* and whether the healthcheck was successful or not.

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 via *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 *compName*.
- No component-invoked healthcheck has been started for the component, designated by *compName,* and the specified *healthcheckKey* parameter.

See Also

saAmfHealthcheckStart(), saAmfComponentErrorReport(), saAmfInitialize()

6.7.4 saAmfHealthcheckStop()

Prototype

```
SaAisErrorT saAmfHealthcheckStop(
        SaAmfHandleT amfHandle,
        const SaNameT *compName,
        const SaAmfHealthcheckKeyT *healthcheckKey
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component for which healthchecks are to be stopped.

healthcheckKey - [in] The key of the healthcheck to be stopped. Using this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters.

Description

This function is used to stop the healthcheck, referred to by *healthcheckKey,* for the component designated by *compName*.

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 *compName*.
- No healthcheck has been started for the component, designated by *compName,* and the specified parameters *healthcheckKey*.

See Also

saAmfHealthcheckStart(), saAmfInitialize()

6.8 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.

6.8.1 saAmfHAStateGet()

Prototype

```
SaAisErrorT saAmfHAStateGet(
        SaAmfHandleT amfHandle,
        const SaNameT *compName,
        const SaNameT *csiName,
        SaAmfHAStateT *haState
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component for which the information is requested.

csiName - [in] A pointer to the name of the component service instance for which the information is requested.

haState - [out] A pointer to the HA state that the Availability Management Framework has currently assigned to the component, identified by *compName,* on behalf of the

component service instance, identified by *csiName*. The HA state is active, standby, quiescing, or quiesced, as defined by the *SaAmfHAStateT* enumeration type.

Description

The Availability Management Framework returns the HA state of a component, identified by *compName,* on behalf of the component service instance, identified by *csiName.*

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 *compName,* has not registered with the Availability Management Framework, or the component has not been assigned the component service instance, identified by *csiName*.

See Also

SaAmfCSISetCallbackT, saAmfInitialize()

6.8.2 SaAmfCSISetCallbackT

Prototype

*typedef void (*SaAmfCSISetCallbackT)(*

SaInvocationT invocation,

*const SaNameT *compName,*

SaAmfHAStateT haState,

SaAmfCSIDescriptorT csiDescriptor

);

Parameters

invocation - [in] This parameter designates a particular invocation of the callback function. The invoked process returns *invocation* when it responds to the Availability Management Framework using the *saAmfResponse()* or *saAmfQuiescingComplete()* functions.

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.

haState - [in] The new HA state to be assumed by the component 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).

csiDescriptor - [in] A pointer to the descriptor with information about the component service instances targeted by this callback invocation.

Description

The Availability Management Framework invokes this callback to request the component, identified by *compName,* to assume a particular 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 the new HA state for the CSI(s) is quiescing, the process must notify the Availability Management Framework when the CSI(s) have been quiesced by using the *saAmfQuiescingComplete()* function. When invoking the *saAmfQuiescingComplete()* function, the process returns *invocation* as an in parameter.

This callback is invoked in the context of a thread of a registered process issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when registering the component, identified by *compName,* via the *saAmfComponentRegister()* call. The Availability Management Framework sets *invocation,* and the process returns *invocation* as an in parameter when it responds to the Availability Management Framework using the *saAmfResponse()* function.
The *saAmfResponse()* function also has *error* as an in parameter, which, in this case, has one of the following possible values:

- SA_AIS_OK - The component executed the *saAmfCSISetCallback()* function successfully.
- SA_AIS_ERR_FAILED_OPERATION - The component failed to assume the HA state, specified by *haState,* for the given component service instance.

If the invoked process does not respond with *saAmfResponse()* within a configured time interval or returns an SA_AIS_ERR_FAILED_OPERATION error value, the Availability Management Framework must engage the configured recovery policy (*recoveryOnError*) for the component to which the process belongs.

See Also

saAmfResponse(), saAmfCSIQuiescingComplete(), saAmfComponentRegister(), saAmfDispatch()

6.8.3 SaAmfCSIRemoveCallbackT

Prototype

```
typedef void (*SaAmfCSIRemoveCallbackT)(
        SaInvocationT invocation,
        const SaNameT *compName,
        const SaNameT *csiName,
        SaAmfCSIFlagsT csiFlags
);
```

Parameters

invocation - [in] This parameter designates a particular invocation of the callback function. The invoked process returns *invocation* when it responds to the Availability Management Framework using the *saAmfResponse()* function.

compName - [in] A pointer to the name of the component from which all component service instances or the component service instance, identified by *csiName,* will be removed.

csiName - [in] A pointer to the name of the component service instance that must be removed from the component identified by *compName.*

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.

Description

The Availability Management Framework requests the component, identified by *compName,* to remove the component service instances identified by *csiName,* from the set of component service instances being supported.

If the value of *csiFlags* is SA_AMF_TARGET_ONE is set, *csiName* contains the component service instance that must be removed. If the value of *csiFlags* is SA_AMF_TARGET_ALL, *csiName* is NULL and the function must remove all component service instances. SA_AMF_TARGET_ALL is always set for components that only provide the “x active or x standby” capability model.

This callback is invoked in the context of a thread of a registered process issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when registering the component, identified by *compName,* via the *saAmfComponentRegister()* call. The Availability Management Framework sets *invocation,* and the component returns *invocation* as an in parameter when it responds to the Availability Management Framework using the *saAmfResponse()* function. The *saAmfResponse()* function also has *error* as an in parameter, which, in this case, has one of the following possible values:

- SA_AIS_OK - The component executed the *saAmfCSIRemoveCallback()* function successfully.
- SA_AIS_ERR_FAILED_OPERATION -The component failed to remove the given component service instance.

If the invoked process does not respond with *saAmfResponse()* within a configured time interval or returns an error then the Availability Management Framework must engage the configured recovery policy (*recoveryOnError*) for the component to which the process belongs.

See Also

saAmfResponse(), saAmfComponentRegister(), saAmfDispatch()

6.8.4 saAmfCSIQuiescingComplete()

Prototype

SaAisErrorT saAmfCSIQuiescingComplete(

SaAmfHandleT amfHandle,

SaInvocationT invocation,

SaAisErrorT error

);

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

invocation - [in] The invocation parameter that the Availability Management Framework assigned when it asked the component to enter the SA_AMF_HA_QUIESCING HA state for a particular component service instance or for all component service instances assigned to it by invoking the *saAmfCSISetCallback()* callback function.

error - [in] The component returns the status of the completion of the quiescing operation, which has one of the following values:

- SA_AIS_OK - The component 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 failed to stop its activity related to a particular component service instance or to all component service instances assigned to it. Some of the actions required during quiescing might not have been performed.

Description

Using this call, a component can notify the Availability Management Framework that it has successfully stopped its activity related to a particular component service instance or to all component service instances assigned to it, following a previous request by the Availability Management Framework, via the component's *SaAmfCSISetCallbackT* callback, to enter the SA_AMF_HA_QUIESCING state for that particular component service instance or to all component service instances.

The invocation of this API indicates that the component has now completed quiescing the particular component service instance or all component service instances and has transitioned to the quiesced HA state for that particular component service instance or to all component service instances.

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 should notify the Availability Management Framework of this fact also using this function. The error parameter specifies whether or not the component has stopped cleanly as requested.

This function may only be called by the registered process of a component, and the *amfHandle* must be the same that was used when the registered process registered this component via the *saAmfComponentRegister*() call.

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).

See Also

SaAmfCSISetCallbackT, saAmfResponse(), saAmfComponentRegister(), *saAmfInitialize()*

6.9 Component Life Cycle

In this section, the callback function to request a component to terminate is described. It contains also additional callback functions that proxy components export to enable the Availability Management Framework to manage proxied components.

6.9.1 SaAmfComponentTerminateCallbackT

Prototype

```
typedef void (*SaAmfComponentTerminateCallbackT)(
        SaInvocationT invocation,
        const SaNameT *compName
);
```

Parameters

invocation - [in] This parameter designates a particular invocation of this callback. The invoked process returns *invocation* when it responds to the Availability Management Framework using the *saAmfResponse()* function.

compName - [in] A pointer to the name of the component to be terminated.

Description

The Availability Management Framework requests the component, identified 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 *compName*.

The component, identified by *compName,* is expected to release all acquired resources and to terminate itself. The invoked process responds by invoking the *saAmfResponse()* function. On return from the *saAmfResponse()* function, the Availability Management Framework removes all service instances associated with the component and the component terminates.

This callback is invoked in the context of a thread of a registered process issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when registering the component, identified by *compName,* via the *saAmfComponentRegister()* call. The component supplies *invocation* and an error code as in parameters to the *saAmfResponse()* function. The error code, in this case, is one of the following values:

- SA_AIS_OK - The function completed successfully.
- SA_AIS_ERR_FAILED_OPERATION - The component identified in *compName* failed to terminate.

If the invoked process does not respond with *saAmfResponse()* within a configured time interval or returns an error then the Availability Management Framework must engage the configured recovery policy (*recoveryOnError*) for the component to which the process belongs.

See Also

saAmfResponse(), saAmfComponentRegister(), saAmfDispatch()

6.9.2 SaAmfProxiedComponentInstantiateCallbackT

Prototype

```
typedef void (*SaAmfProxiedComponentInstantiateCallbackT)(
        SaInvocationT invocation,
        const SaNameT *proxiedCompName
);
```

Parameters

invocation - [in] This parameter designates a particular invocation of this callback function. The invoked process returns *invocation* when it responds to the Availability Management Framework using the *saAmfResponse()* function.

proxiedCompName - [in] A pointer to the name of the proxied component to be instantiated.

Description

The Availability Management Framework requests a proxy component to instantiate a proxied component, identified by *proxiedCompName*. The proxy component to which this request is addressed must have registered the proxied component with the Availability Management Framework before the Availability Management Framework invokes this function.

This callback is invoked in the context of a thread of a registered process for a proxy component issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when registering the component, identified by *proxiedCompName,* via the *saAmfComponentRegister()* call. The invoked process responds by invoking the *saAmfResponse()* function, supplying *invocation* and *error* as in parameters; in this case, *error* has 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 might issue a further attempt to instantiate the proxied component.

If the invoked process does not respond with *saAmfResponse()* within a configured time interval or returns an SA_AIS_ERR_FAILED_OPERATION error value, the Availability Management Framework must engage the configured recovery policy (*recoveryOnError*) for the component to which the process belongs.

See Also

saAmfResponse(), saAmfComponentRegister(), saAmfDispatch(), SaAmfProxiedComponentCleanupCallbackT

6.9.3 SaAmfProxiedComponentCleanupCallbackT

Prototype

```
typedef void (*SaAmfProxiedComponentCleanupCallbackT)(
        SaInvocationT invocation,
        const SaNameT *proxiedCompName
);
```

Parameters

invocation - [in] This parameter designates a particular invocation of this callback function. The invoked process returns *invocation* when it responds to the Availability Management Framework using the *saAmfResponse()* function.

proxiedCompName - [in] A pointer to the name of the proxied component to be abruptly terminated.

Description

The Availability Management Framework requests a proxy component to abruptly terminate a proxied component, identified by *proxiedCompName*. The proxy component to which this request is addressed must have registered with the Availability Management Framework before the Availability Management Framework invokes this function.

This callback is invoked in the context of a thread of a registered process for a proxy component issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when registering the component, identified by *proxiedCompName,* via the *saAmfComponentRegister()* call. The invoked process responds by invoking the *saAmfResponse()* function, supplying *invocation* and *error* as in parameters; in this case, *error* has 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.

If the invoked process does not respond with *saAmfResponse()* within a configured time interval or returns an SA_AIS_ERR_FAILED_OPERATION error value, the Availability Management Framework must engage the configured recovery policy (*recoveryOnError*) for the component to which the process belongs.

See Also

saAmfResponse(), saAmfComponentRegister(), saAmfDispatch(), SaAmfProxiedComponentInstantiateCallbackT

6.10 Protection Group Management

6.10.1 saAmfProtectionGroupTrack()

Prototype

SaAisErrorT saAmfProtectionGroupTrack(

SaAmfHandleT amfHandle,

*const SaNameT *csiName,*

SaUint8T trackFlags,

*SaAmfProtectionGroupNotificationBufferT *notificationBuffer*

);

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

csiName - [in] A pointer to the name of the component service instance, which is also the name of the protection group, for which tracking is to start.

trackFlags - [in] The kind of tracking that is requested, which is the bitwise OR of one or more of the flags SA_TRACK_CURRENT, SA_TRACK_CHANGES or SA_TRACK_CHANGES_ONLY, defined in the SA Forum Overview document, 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 *notificationBuffer* when the *saAmfProtectionGroupTrack()* call completes.

- 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 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 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.

notificationBuffer - [in/out] - A pointer to a buffer of type *SaAmfProtectionGroupNotificationBufferT*. This parameter is ignored if SA_TRACK_CURRENT is not set in *trackFlags*; otherwise, if *notificationBuffer* is not NULL, the buffer will contain information about all components in the protection group when *saAmfProtectionGroupTrack()* returns. The meaning of the fields of the *SaAmfProtectionGroupNotificationBufferT* buffer is:

- *numberOfItems* - [in/out] If *notification* is NULL, *numberOfItems* is ignored as input parameter; otherwise, it specifies that the buffer pointed to by *notification* provides memory for information about *numberOfItems* components in the protection group.
  When *saAmfProtectionGroupTrack()* 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()* function.

Description

The Availability Management Framework is requested to start tracking changes in the protection group associated with the component service instance, identified by *csiName,* or changes of attributes of any component in the protection group. These changes are notified via the invocation of the *saAmfProtectionGroupTrackCallback()* callback function, which must have been supplied when the process invoked the *saAmfInitialize()* call.

An application may call *saAmfProtectionGroupTrack()* repeatedly for the same values of *amfHandle* and *csiName,* regardless of whether the call initiates a one-time status request or a series of callback notifications. If *saAmfProtectionGroupTrack()* is called with *trackFlags* containing SA_TRACK_CHANGES_ONLY, while changes in the pro-

tection group are currently being tracked with SA_TRACK_CHANGES for the same combination of *amfHandle* and *csiName*, the Availability Management Framework will invoke further notification callbacks according to the new *trackFlags*. The same is true vice versa.
Once *saAmfProtectionGroupTrack()* 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 initialization with *saAmfInitialize()* was incomplete, since the *saAmfProtectionGroupTrackCallback()* callback function is missing. This value is not returned if *trackFlags* is set to SA_TRACK_CURRENT and the *notificationBuffer* 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* field in *notificationBuffer* is not NULL, but the *numberOfItems* field in *notificationBuffer* indicates that the provided buffer is too small to hold information about all components in the protection group.

The *numberOfItems* field in *notificationBuffer* indicates that the provided buffer is too small to hold information about all components in the protection group.

SA_AIS_ERR_NOT_EXIST - The component service instance, designated by *csiName,* cannot be found.

SA_AIS_ERR_BAD_FLAGS - The *trackFlags* parameter is invalid.

See Also

SaAmfProtectionGroupTrackCallbackT, saAmfProtectionGroupTrackStop(), saAmfProtectionGroupNotificationFree(), saAmfInitialize()

6.10.2 SaAmfProtectionGroupTrackCallbackT

Prototype

```
typedef void (*SaAmfProtectionGroupTrackCallbackT)(
        const SaNameT *csiName,
        SaAmfProtectionGroupNotificationBufferT *notificationBuffer,
        SaUint32T numberOfMembers,
        SaAisErrorT error
);
```

Parameters

csiName - [in] A pointer to the name of the component service instance.

notificationBuffer - [in] A pointer to a notification buffer, which contains the requested information about components in the protection group.

numberOfMembers - [in] The number of the components that belong to the protection group associated with the component service instance, designated by *csiName*.

error - [in] This parameter indicates whether the Availability Management Framework was able to perform the operation. The parameter *error* has one of the 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 the *saAmfProtectionGroupTrack()* call later.
- SA_AIS_ERR_BAD_HANDLE - The handle *amfHandle* that was passed to the corresponding *saAmfProtectionGroupTrack()* call is invalid, since it is corrupted, uninitialized, or has already been finalized.
- SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly in the corresponding *saAmfProtectionGroupTrack()* call.

- 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()* 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()* might have missed one or more notifications.
- SA_AIS_ERR_NOT_EXIST - The component service instance, designated by the parameter *csiName,* has been administratively deleted.
- SA_AIS_ERR_BAD_FLAGS - The *trackFlags* parameter is invalid in the corresponding *saAmfProtectionGroupTrack()* call.

If the error returned is SA_AIS_ERR_NO_MEMORY or SA_AIS_ERR_NO_RESOURCES, the process that invoked *saAmfProtectionGroupTrack()* should invoke *saAmfProtectionGroupTrackStop()* and then invoke *saAmfProtectionGroupTrack()* again to resynchronize with the current state.

Description

This callback is invoked in the context of a thread issuing an *saAmfDispatch()* call on the handle *amfHandle,* which was specified when the process requested tracking of changes in the protection group associated with the component service instance, identified by *csiName,* or in an attribute of any component in this protection group via the *saAmfProtectionGroupTrack()* call. If successful, the *saAmfProtectionGroupTrackCallback()* function returns the requested information in the *notificationBuffer* parameter. The kind of information returned depends on the setting of the *trackFlags* parameter of the *saAmfProtectionGroupTrack()* function.

The value of the *numberOfItems* attribute in the *notificationBuffer* parameter 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 *notificationBuffer* 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(), saAmfProtectionGroupTrackStop(), saAmfDispatch()

6.10.3 saAmfProtectionGroupTrackStop()

Prototype

```
SaAisErrorT saAmfProtectionGroupTrackStop(
        SaAmfHandleT amfHandle,
        const SaNameT *csiName
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

csiName - [in] A pointer to the name of the component service instance.

Description

The invoking process requests the Availability Management Framework to stop tracking protection group changes for the component service instance designated by *csiName*.

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 *csiName,* cannot be found,
- No track of protection group changes for *csiName* was previously started via the *saAmfProtectionGroupTrack()* call with track flags SA_TRACK_CHANGES or SA_TRACK_CHANGES_ONLY, and which is still in effect.

See Also

SaAmfProtectionGroupTrackCallbackT, saAmfProtectionGroupTrack(), saAmfInitialize()

6.10.4 saAmfProtectionGroupNotificationFree()

Prototype

SaAisErrorT saAmfProtectionGroupNotificationFree(

SaAmfHandleT amfHandle,

*SaAmfProtectionGroupNotificationT *notification*

);

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating this particular initialization of the Availability Management Framework.

notification - [in] A pointer to the notification buffer that was allocated by the Availability Management Framework library in the *saAmfProtectionGroupTrack()* function and is to be released.

Description

This function frees the memory pointed to by *notification* and that was allocated by the Availability Management Framework library in a previous call to the *saAmfProtectionGroupTrack*() function.

For details, refer to the *notificationBuffer* parameter in the corresponding invocation of the *saAmfProtectionGroupTrack()* 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.

See Also

saAmfProtectionGroupTrack()

6.11 Error Reporting

6.11.1 saAmfComponentErrorReport()

Prototype

```
SaAisErrorT saAmfComponentErrorReport(
    SaAmfHandleT amfHandle,
    const SaNameT *erroneousComponent,
    SaTimeT errorDetectionTime,
    SaAmfRecommendedRecoveryT recommendedRecovery,
    SaNtfIdentifierT ntfIdentifier
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

erroneousComponent - [in] A pointer to the name of the erroneous component.

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.

recommendedRecovery - [in] Recommended recovery action.

ntfIdentifier - [in] Identifier of the notification sent by the component to the Notification Service (see [2]) prior to reporting the error to the Availability Management Framework.

Description

The *saAmfComponentErrorReport()* 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 3.12.2.1 on page 141.

Prior to reporting the error to the Availability Management Framework, the component should send a notification to the Notification Service providing adequate information for cause analysis. The notification identifier returned by the Notification Service must be provided in the *ntfIdentifier* parameter for correlation purposes. In the case where no notification is produced prior to this call, the special value SA_NTF_IDENTIFIER_UNUSED (see [2]) is passed in *ntfIdentifier*.

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.

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 *compName*, is not contained in the Availability Management Framework's configuration.

SA_AIS_ERR_ACCESS - The Availability Management rejects the requested recommended recovery.

See Also

saAmfComponentErrorClear(), saAmfInitialize()

6.11.2 saAmfComponentErrorClear()

Prototype

SaAisErrorT saAmfComponentErrorClear(

SaAmfHandleT amfHandle,

*const SaNameT *compName,*

SaNtfIdentifierT ntfIdentifier

);

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

compName - [in] A pointer to the name of the component to be cleared of any error.

ntfIdentifier - [in] Identifier of the notification sent by the component to the Notification Service (see [2]).

Description

The function cancels the previous errors reported about the component, identified by *compName*. The Availability Management Framework may now change the component's operational state to "enabled", assuming that nothing else prevents this. The Availability Management Framework may, then, perform additional assignments of component service instances to the component.

Before clearing all errors reported about the component, a notification should be sent by the component to the Notification Service providing adequate information to properly clear active alarms. The notification identifier returned by the Notification Service must be provided in the *ntfIdentifier* parameter for correlation purposes. In the case where no notification is produced prior to this call, the special value SA_NTF_IDENTIFIER_UNUSED (see [2]) is passed in *ntfIdentifier.*

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 *compName*, is not contained in the Availability Management Framework's configuration.

See Also

saAmfComponentErrorReport(), saAmfInitialize()

6.12 Component Response to Framework Requests

6.12.1 saAmfResponse()

Prototype

```
SaAisErrorT saAmfResponse(
        SaAmfHandleT amfHandle,
        SaInvocationT invocation,
        SaAisErrorT error
);
```

Parameters

amfHandle - [in] The handle, obtained through the *saAmfInitialize()* function, designating a particular initialization of the Availability Management Framework.

invocation - [in] This parameter associates an invocation of this response function with a particular invocation of a callback function by the Availability Management Framework.

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.

Description

The component responds to the Availability Management Framework with the result of its execution of a particular request of the Availability Management Framework, designated by *invocation*. The request can be of one of the following types:

- Request for executing a given healthcheck. See *saAmfHealthcheckCallback()*.
- Request for terminating a component. See *saAmfComponentTerminateCallbackT*.
- Request for adding/assigning a given 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*.

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.

With the exception of the response to an *saAmfHealthcheckCallback()* call, this function may be called only by a registered process, that is, the *amfHandle* must be the same that was used when the registered process registered this component via the *saAmfComponentRegister()* call. 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).

See Also

SaAmfHealthcheckCallbackT, SaAmfComponentTerminateCallbackT, SaAmfCSISetCallbackT, SaAmfCSIRemoveCallbackT, SaAmfProxiedComponentInstantiateCallbackT, SaAmfProxiedComponentCleanupCallbackT, saAmfComponentRegister(), saAmfInitialize()

7 Administrative API

7.1 Availability Management Framework Administration API Model

7.1.1 Availability Management Framework Administration API Basics

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, SNMP agents and CIM providers that typically convert system administration commands (invoked from a management station) to the correct administrative API sequence to yield the desired result that is expected upon execution of the system administration command.

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 a node using an Availability Management Framework administration API shall restart all the components contained in the service units housed in the node. This operation will not reboot the node. Similarly, restarting a cluster in the context of Availability Management Framework shall restart all components in the cluster, but shall not reboot the nodes in the 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) or a node. In certain cases, an exception has been made, where the administration operation directly affects a component within a service unit; however, those exceptions are rare. This choice of granularity for administrative operations aligns with Section 3.2.4, which advocates a coarser-grained and aggregated view of the components via the service unit to the system administrator.

Administrative operations that are applicable to the lowest granular logical entity are called the **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 desired result. These operations are called **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 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, we consider that concurrent and potentially conflicting administrative operations are invalid, i.e., when an administrator has initiated an administrative operation on a logical entity 'A', any other administrative operation that

involves a logical entity with which this logical entity ‘A’ has a relationship (association or aggregation) will not be allowed until the first operation on ‘A’ is done.

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 node-level and then unlock each service unit one by one at the service unit-level. This 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 [4]).

7.2 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 [4]).

To use the Availability Management Framework administration API, an application must be bound to the IMM Service library (see [4] for the library name).

7.3 Type Definitions

The specification of Availability Management Framework Administration API requires the following types, in addition to the ones already described.

7.3.1 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;

7.4 Availability Management Framework Administration API

As explained above, the administrative API shall be exposed by the IMM Service library. The IMM Service API *saImmOmAdminOperationInvoke()* or *saImmOmAdminOperationInvokeAsync()* functions shall be invoked with the appropriate *operationId* (see Section 7.3.1) and *objectName* to execute a particular administrative operation. In the following section, the administrative APIs are described with

the assumption that the Availability Management Framework is an object implementer for the various administrative operations that will be initiated as a consequence of invoking the *saImmOmAdminOperationInvoke()* or *saImmOmAdminOperationInvokeAsync()* functions (see [4]) with the appropriate *operationId* (described in Section 7.3.1) on the entity designated by *objectName*.

The API syntax for the administrative APIs shall only use the corresponding enumeration value for the *operationId* as explained in 7.3.1 on page 231 for administrative operations on various Availability Management Framework logical entities along with *objectName* and the possible return values.

The return values explained in the sections below for various administrative operations shall be passed in the *operationReturnValue* parameter, which is provided by the invoker of the *saImmOmAdminOperationInvoke()* or *saImmOmAdminOperationInvokeAsync()* functions to obtain return codes from the object implementer (Availability Management Framework, in this case).

The operations described below are applicable to and have the same effects on both pre-instantiable and non-pre-instantiable service units, unless explicitly stated otherwise.

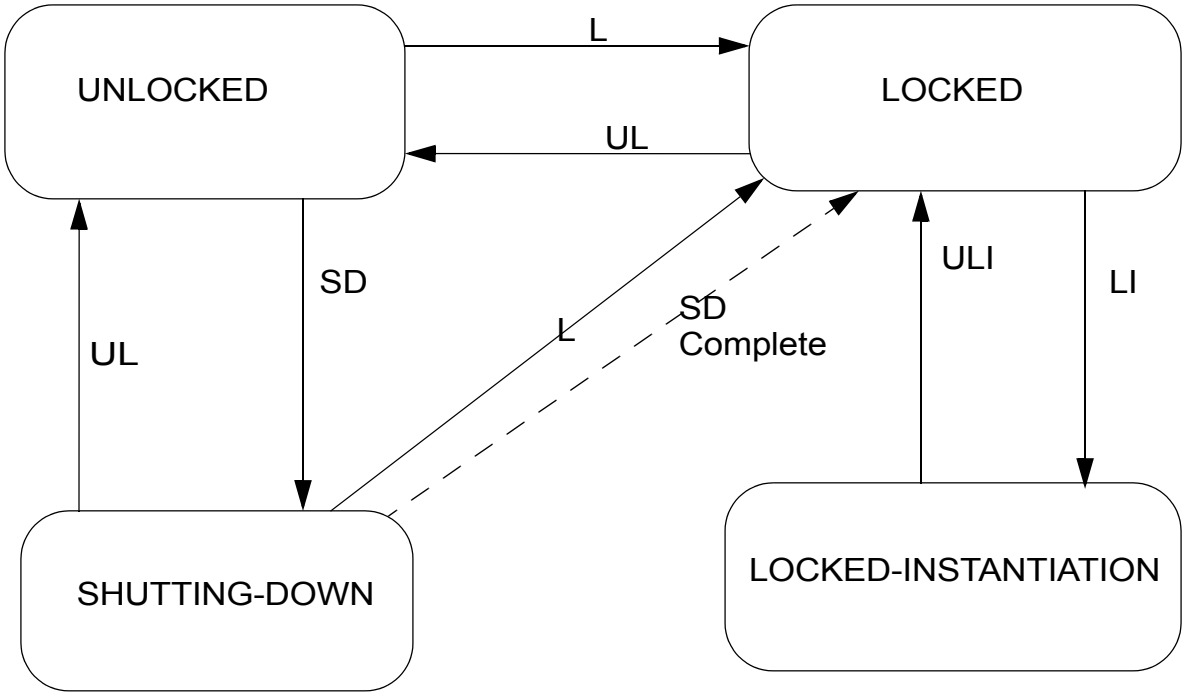
7.4.1 Administrative State Modification Operations

A fair number of administrative operations involve the manipulation of the administrative state. In order to aid in the description of such administrative operations, a figure is provided illustrating the various administrative states and the various operations that are applicable on an entity when it is in a particular administrative state. The figure below uses abbreviated form for the description of administrative operations and these correspond to the following:

- 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 that transitions the entity into locked state without further external intervention.

FIGURE 26 Administrative States and Related Operations for Availability Management Framework Entities



7.4.2 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, node, a service group, an application, and the cluster, i.e., all entities that possess an administrative state.

The invocation of this administrative operation sets the administrative state of the logical entity designated by *objectName* to unlocked. Refer to Sections 3.3.1.2 on page 41 (service unit), 3.3.3.1 on page 56 (SI), 3.3.5 on page 58 (service group), 3.3.6.1 on page 59 (node), 3.3.7 on page 61 (application) and 3.3.8 on page 61 (cluster) for more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities.

If this operation is invoked on an entity that is already unlocked, there is no change in the status of such an entity, i.e., 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 an entity that is locked for instantiation, there is no change in the status of such an entity, i.e., 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative procedure is not supported by the type of entity denoted by *objectName*.

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*

7.4.3 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, a node, a service group, an application, and the cluster, i.e., all Availability Management Framework entities that support an administrative state.

The invocation of this administrative operation sets the administrative state of the logical entity designated by *objectName* to locked. Refer to Sections 3.3.1.2 on page 41 (service unit), 3.3.3.1 on page 56 (SI), 3.3.5 on page 58 (service group), 3.3.6.1 on page 59 (node), 3.3.7 on page 61 (application) and 3.3.8 on page 61 (cluster) for more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities.

If this operation is invoked by a client on an entity that is already locked, there is no change in the status of such an entity, i.e., it remains in the locked state, but a benign error value SA_AIS_ERR_NO_OP is returned to the client conveying that the entity in question, designated by *objectName,* is already in locked state.

If this operation is invoked on an entity that is locked for instantiation, there is no change in the status of such an entity, i.e., 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED -This administrative procedure is not supported by the type of entity denoted by *objectName*.

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 co-operate 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

7.4.4 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, a node, a service group, an application, and the cluster.

The invocation of this administrative operation sets the administrative state of the logical entity designated by *objectName* to locked-instantiation subject to constraints described below, causing all relevant service units to become non-instantiable after their termination. Refer to Sections 3.3.1.2 on page 41 (service unit), 3.3.5 on page 58 (service group), 3.3.6.1 on page 59 (node), 3.3.7 on page 61 (application) and

3.3.8 on page 61 (cluster) for more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities.

After successful invocation of this procedure, all components in all pertinent service units are terminated; in particular, all processes in those components must cease to exist.

Once this operation is invoked on a logical entity, as explained above, all pertinent service units within its scope become non-instantiable (after being terminated) and 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 mode 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.

If the entity is unavailable (for example if a node is configured but not a member) during the invocation of this administrative operation, all service units within the scope of the entity are set to non-instantiable and they can only be ever again instantiated in a locked state after another administrative operation designated by the *operationId* SA_AMF_ADMIN_UNLOCK_INSTANTIATION (refer to Section 7.4.5 on page 238) 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 an entity that is already in locked-instantiation state, there is no change in the status of such an entity, i.e., it remains in that state but 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 an entity that is either in the shutting-down or unlocked administrative state, there is no change in the status of such an entity, i.e., it 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED -This administrative procedure is not supported by the type of entity denoted by *objectName*.

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 co-operate 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_UNLOCK_INSTANTIATION

7.4.5 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, a node, a service group, an application, and the cluster, i.e., all Availability Management Framework entities that support an administrative state with a locked-instantiation value.

The invocation of this administrative operation sets the administrative state of the logical entity designated by *objectName* to locked. Refer to Sections 3.3.1.2 on page 41 (service unit), 3.3.3.1 on page 56 (SI), 3.3.5 on page 58 (service group), 3.3.6.1 on

page 59 (node), 3.3.7 on page 61 (application) and 3.3.8 on page 61 (cluster) for more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities.

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 locked state), provided that the concerned service units are not locked for instantiation at some other level. A subsequent invocation of the *SA_AMF_ADMIN_UNLOCK* administrative operation would make the relevant service units available for SI assignment by the Availability Management Framework.

If this operation is invoked by a client on an entity that is already locked, there is no change in the status of such an entity, i.e., it remains in the locked state, but a benign error value SA_AIS_ERR_NO_OP is returned to the client conveying that the entity in question, designated by *objectName,* is already in locked state.

If this operation is invoked by a client on an entity that is either in the shutting-down or unlocked administrative state, there is no change in the status of such an entity, i.e., it 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED -This administrative procedure is not supported by the type of entity denoted by *objectName*.

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

7.4.6 SA_AMF_ADMIN_SHUTDOWN

Parameters

operationId = *SA_AMF_ADMIN_SHUT_DOWN*

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, a node, a service group, an application, and the cluster, i.e., all Availability Management Framework entities that support an administrative state.

The invocation of this administrative operation sets the administrative state of the logical entity designated by *objectName* to shutting-down. This administrative operation is non-blocking, i.e., it does not wait for the logical entity designated by *objectName* to transition to the locked state, which can possibly take a very long time. Refer to Sections 3.3.1.2 on page 41 (service unit), 3.3.3.1 on page 56 (SI), 3.3.5 on page 58 (service group), 3.3.6.1 on page 59 (node), 3.3.7 on page 61 (application) and 3.3.8 on page 61 (cluster) for more details regarding the respective status of the logical entities that results as a consequence of invoking this administrative operation on these entities.

If this operation is invoked on an entity that is already in shutting-down administrative state, there is no change in the status of such an entity, i.e., 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 an entity that is either in locked or locked-instantiation administrative state, there is no change in the status of such an entity, i.e., it remains locked or locked for instantiation, 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative procedure is not supported by the type of entity denoted by *objectName*.

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 co-operate 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

7.4.7 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, a node, an application, and the 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 3.12.1.2 on page 135.

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', reassigning the assigned service instances is not necessary, but if at-least one component within the service unit has the configuration option of *disableRestart* set to TRUE, then 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 on an individual instantiated component, only the component implied in the operation is restarted. If the component in question has the configuration option of *disableRestart* set to TRUE, then, depending upon the redundancy model, a reassignment of the service instances assigned to the service unit to which the component belongs shall be undertaken by the Availability Management Framework. In other words, restarting such a component will potentially affect the entire service unit to which it belongs.

When invoked upon a node, an application or the cluster, this action becomes a composite operation that causes a collective restart of all service units residing within the

node, application or the cluster. In-order 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 instantiating them back until all service units have been terminated. In the cases of application restart and cluster restart, 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 cluster.

The Availability Management Framework may 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, an error value of SA_AIS_ERR_TRY_AGAIN should be returned indicating that the action is feasible but not at this instant.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_BAD_OPERATION - The target logical entity for this operation, identified by *objectName* 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 procedure is not supported by the type of entity denoted by *objectName*.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not co-operate 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

7.4.8 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 pertinent 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.

If the SI designated by *objectName* 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 this SI but any other SI that is assigned to a service unit to which the SI designated by *objectName* is assigned. Note that this behavior is consistent with the semantics of the respective redundancy model.

If the SI designated by *objectName* 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 this SI but any other SI that is assigned active to a service unit to which the SI designated by *objectName* is assigned active. Application of this operation on a SI may potentially modify the standby assignments of other SIs which 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 14 on page 85: If the swap operation is applied on SI A, then 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 3 (on Node W) and Service Unit 2 (on Node V) respectively.

In case 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 redundancy model. It is possible that in N-Way redundancy model a SI has multiple standby assignments in which case this administrative operation shall affect only the highest ranked standby assignment.

This operation may not be invoked on a SI, which 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 cluster is in a degenerated status and reduction procedures have been engaged) when this operation is invoked on a particular logical entity, an error value SA_AIS_ERR_FAILED_OPERATION 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 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 between 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_BAD_OPERATION - The operation was not successful on the target SI, possibly because no standby assignments are available for the SI or the service

group protecting the SI has either a No Redundancy or N-Way Active redundancy model.

SA_AIS_ERR_NOT_SUPPORTED - This administrative procedure is not supported for the type of entity denoted by *objectName*.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not co-operate 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

7.4.9 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 relevant 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, the best possible configuration will be restored where the rankings of the service units are respected with regards to active and standby SI assignments.

The objective of this administrative operation is to provide an administrator the capability to manually execute an adjust procedure as described in Section 3.7.1.1 on page 71. This command is generally issued after the service group has undergone a series of swaps, locks or shut-downs, 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 and 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative procedure is not supported for the type of entity denoted by *objectName*.

SA_AIS_ERR_REPAIR_PENDING - If during the execution of this operation, certain erroneous components do not co-operate 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

7.4.10 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 a node.

This administrative operation is used to clear the disabled operational state of a node or a service unit after they have been successfully mended to declare them as repaired. The administrator uses this command to indicate the availability of a service unit or a node for providing service after an externally executed repair action. When invoked on a node, this operation results 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. A node or a service unit enters the disabled operational state due to reasons stated in Section 3.3.6.2 on page 59 (node) and 3.3.1.3 on page 42 (service unit).

The Availability Management Framework might optionally engage in repairing a 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 node or service unit, but 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 node or the service unit to indicate that these entities are repaired and their operational state is enabled.

It is expected that repair done by an external entity should bring the repaired service units and components in a consistent state, i.e., to either instantiated or uninstantiated presence state and this procedure should ensure that this is indeed the case before an SA_AIS_OK status is returned by this operation.

If this administrative operation is invoked on a cluster 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_SUPPORTED - This administrative procedure is not supported by the type of entity denoted by *objectName*.

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

7.4.11 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 procedure 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_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error should be generally returned in cases where 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_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 procedure is not supported for the type of entity denoted by *objectName*.

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

7.4.12 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 procedure 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 should be generally returned in cases where 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_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 procedure is not supported for the type of entity denoted by *objectName*.

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

7.5 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 17 Summary: Applicability of Administrative Operations

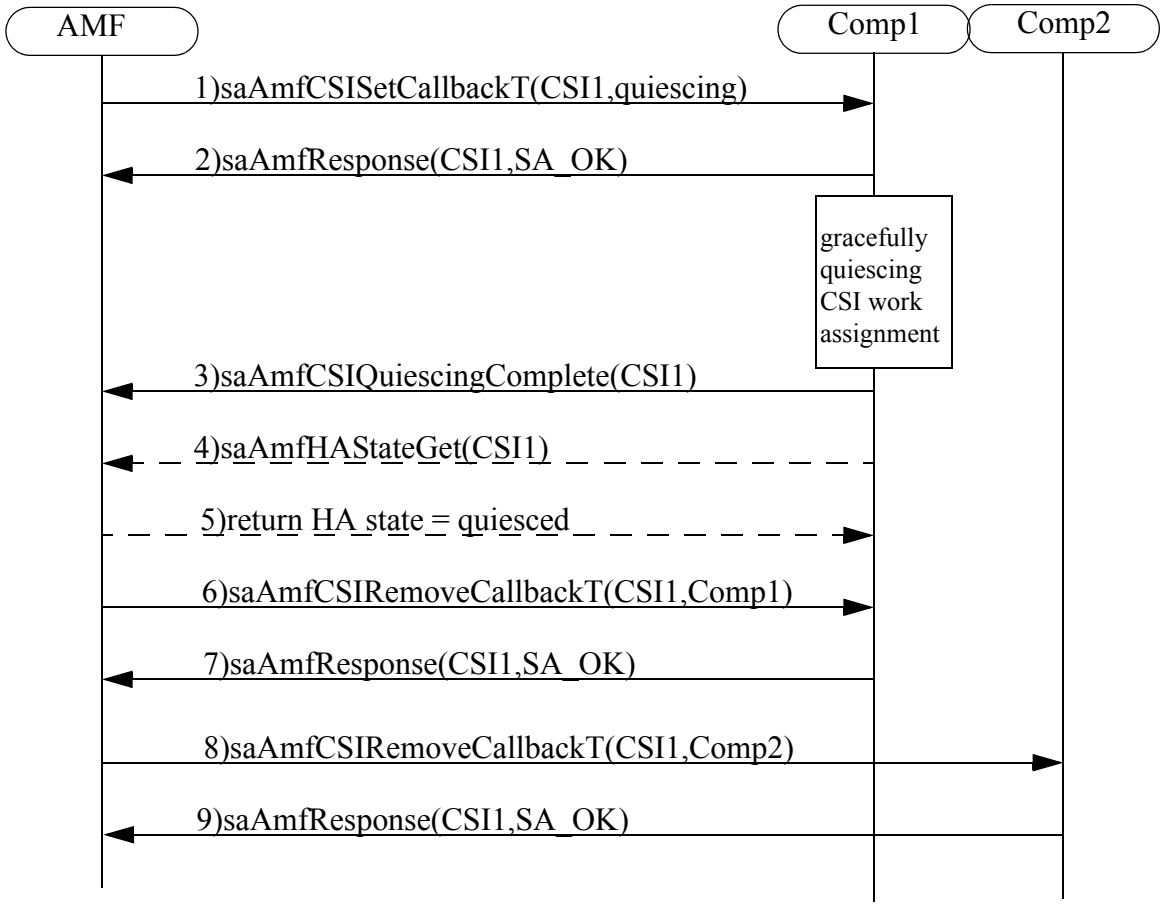
Administrative Operation	Applicability
UNLOCK	cluster, application, SG, node, SU, SI
LOCK	cluster, application, SG, node, SU, SI
UNLOCK_INSTANTIATION	cluster, application, SG, node, SU
LOCK_INSTANTIATION	cluster, application, SG, node, SU
SHUT_DOWN	cluster, application, SG, node, SU, SI
RESTART	cluster, application, node, SU, component
SWAP_SI	SI
ADJUST_SG	SG
REPAIRED	node, SU
EAM_START	SU, component
EAM_STOP	SU, component

8 Basic Operational Scenarios

The following sequence diagrams describe basic operational scenarios.

8.1 Administrative Shutdown of a Service Instance

The context of this scenario is a service group with 2N redundancy model having two service units with a single SA-aware component each. Two SIs are assigned to the service unit such that component 1 (Comp1) and component 2 (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 when one of the two SIs is administratively shut down.



The dotted lines indicate optional transactions.

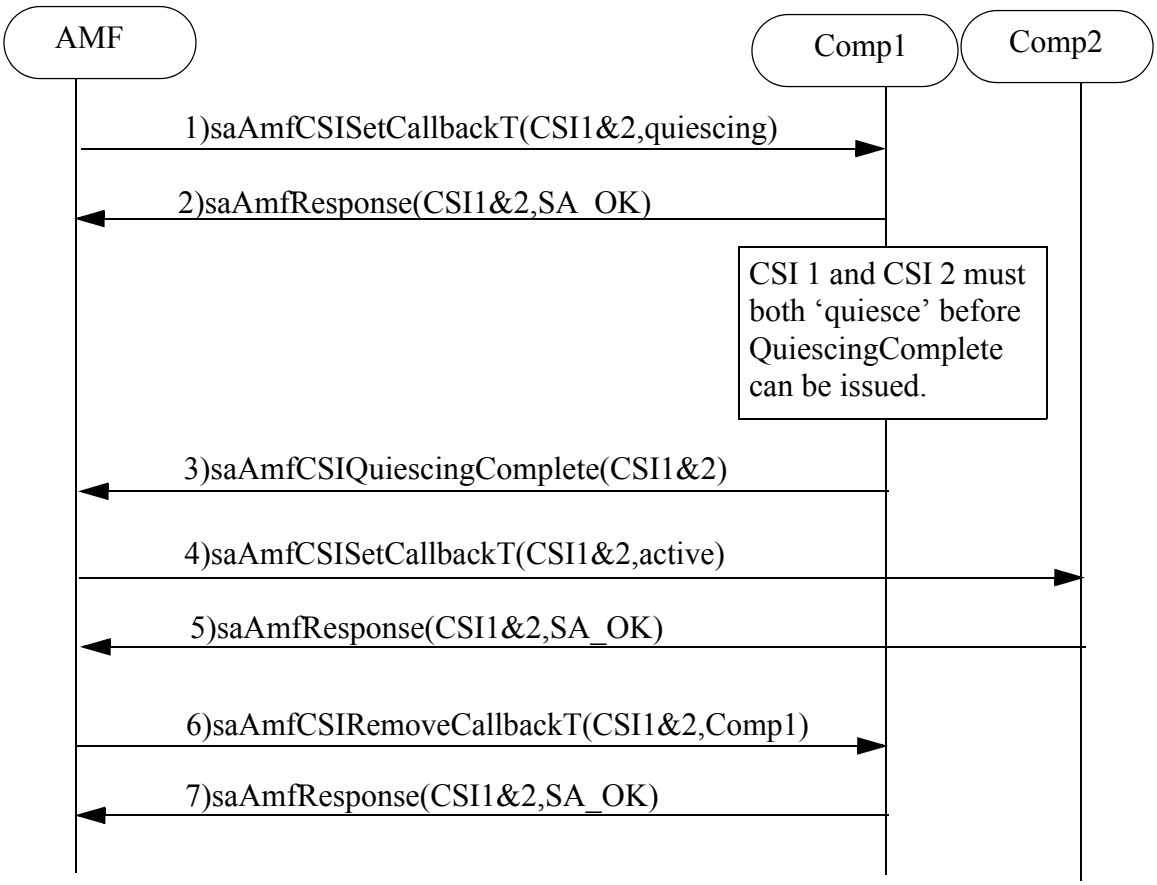
The result of a “complete” transition from the quiescing HA state is to arrive at the quiesced HA state.

Notice that since 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 alone.

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.

8.2 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 SA-aware component each. Two SIs are assigned to the service unit such that component 1 (Comp1) and component 2 (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 when one of the service units (the one having components assigned active for CSI1 and CSI2) is administratively shut down.



The Availability Management Framework should use *csiFlags* value SA_AMF_TARGET_ALL in (callback) steps 1, 4 and 6 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".

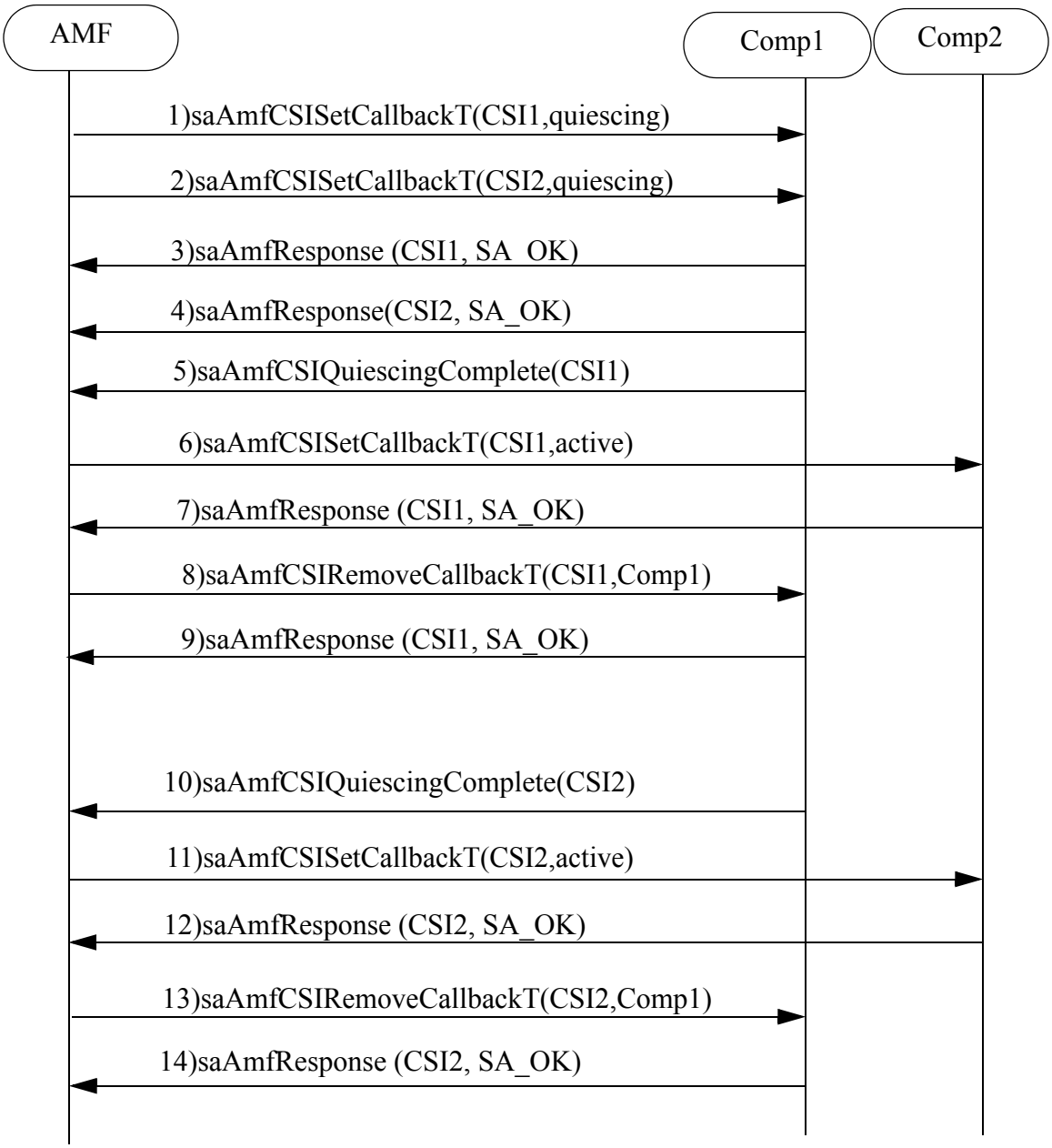
Notice that *saAmfCSIQuiescingComplete()* can only be invoked when all component service instance assignments have successfully quiesced within the component.

8.3 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 SA-aware component each. Two SIs are assigned to the service units such that component 1 (Comp1) and component 2 (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 when one of the service units (the one having components assigned active for CSI1 and CSI2) is administratively shut down.

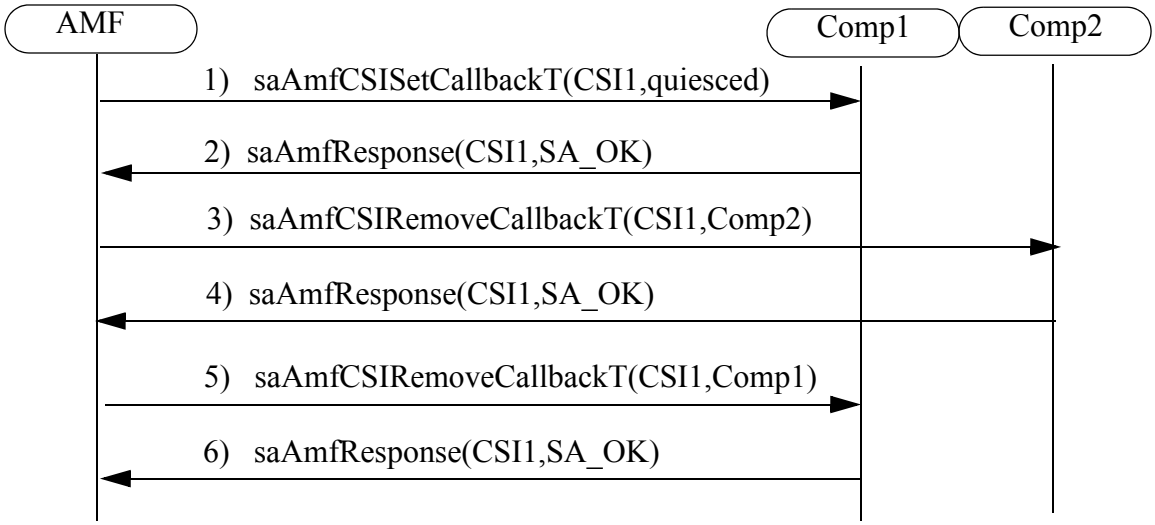


Note that Comp2 will have both active and standby assignments for a certain period of time, which implies that Comp2 must have the X_active_and_Y_standby capability.

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.

8.4 Administrative Locking of a Service Instance

The context of this scenario is a service group with 2N redundancy model having two service units with a single SA-aware component. Two SIs are assigned to the service units such that component 1 (Comp1) and component 2 (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 when one of the two SIs are locked.

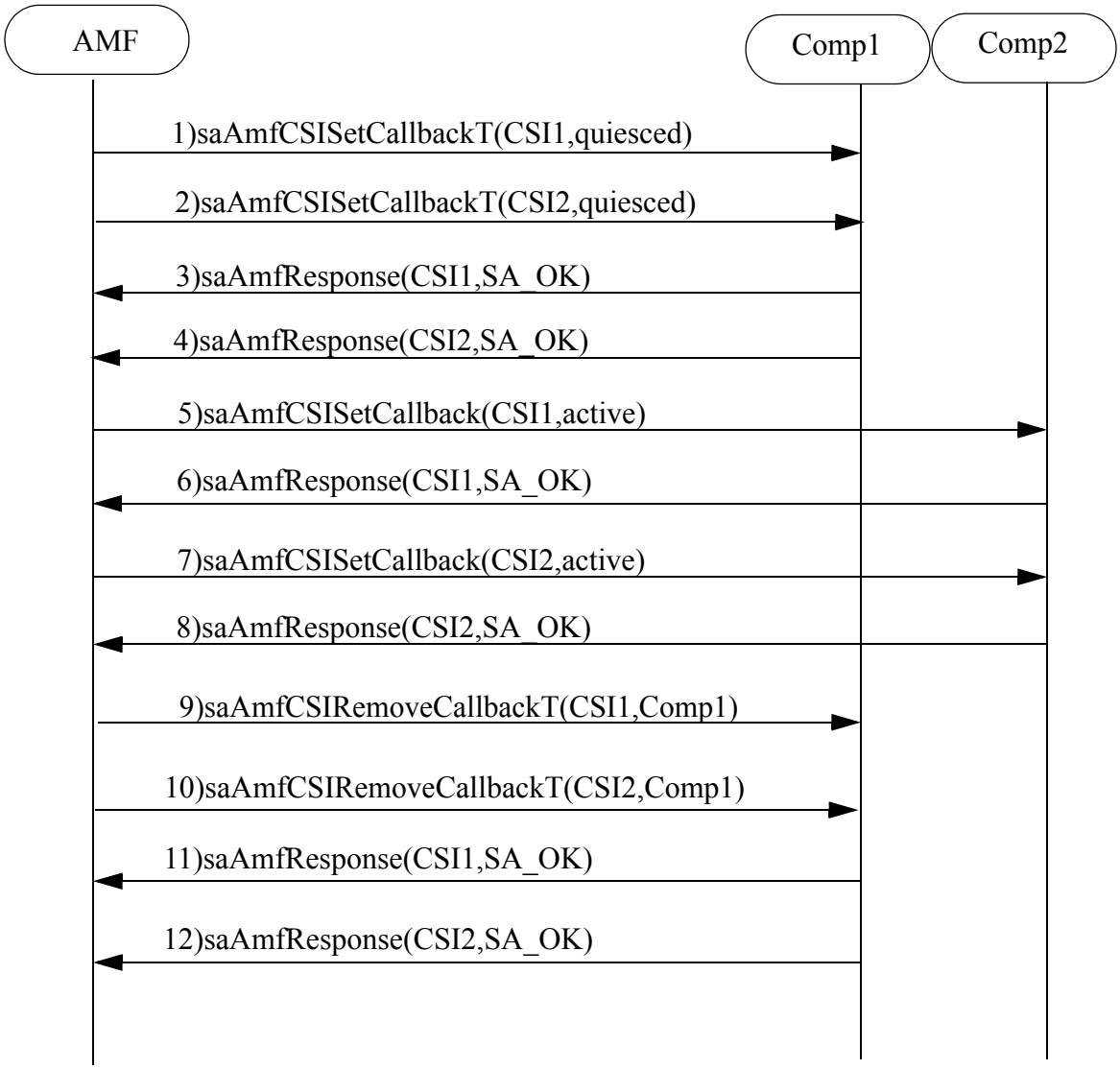


Notice that since only one of the SIs have been locked, the component service instance corresponding to that SI is manipulated.

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.

8.5 Administrative Locking of a Service Unit

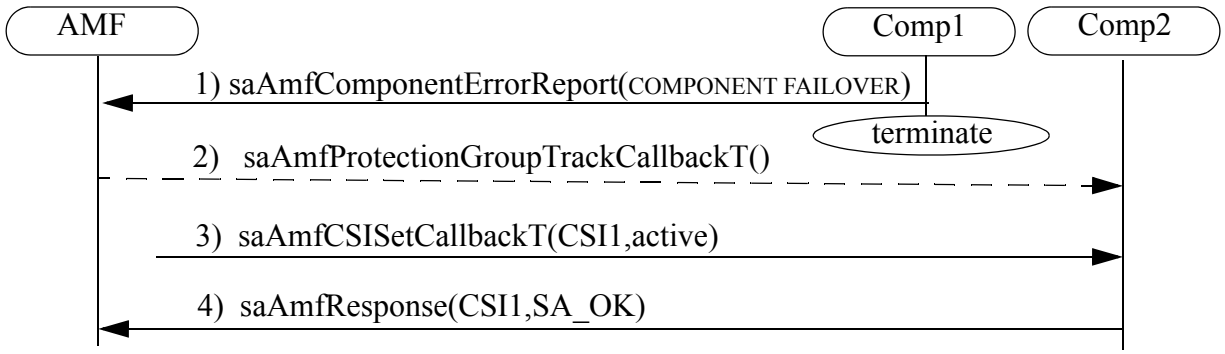
The context of this scenario is a service group with 2N redundancy model having two service units with a single SA-aware component each. Two SIs are assigned to the two service units such that component 1 (Comp1) and component 2 (Comp2) have each two CSI assignments: Comp1 is assigned active for CSI1 and CSI2, and Comp2 is assigned standby for CSI1 and CSI2. The following diagram shows the sequence when one of the service units (the one having components assigned active for CSI1 and CSI2) is administratively locked.



Note that the same sequence 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.

8.6 A Simple Fail-Over

The context of this scenario is a service group with a 2N redundancy model having two service units, each with a single SA-aware component. A single SI is assigned such that component 1 (Comp1) and component 2 (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.



The dotted line indicates an optional transaction. Note that the protection group callback informs the registered component that Comp1 exited from the protection group.

9 Alarms and Notifications

The Availability Management Framework produces certain alarms and notifications in order to convey important information regarding its operational and functional state to an administrator or a management system.

These reports vary in perceived severity and include alarms, which potentially require an operator intervention and notifications that signify important state or object changes. A management entity should regard notifications, but they do not necessarily require an operator intervention.

The recommended vehicle to be used for producing alarms and notifications is the Notification Service of the Service Availability™ Forum (abbreviated as NTF, see [2]), 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, which the vendor is at a 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 [6]).

9.1 Setting Common Attributes

The tables presented in Section 9.2 refer to the attributes in the following list, but do not describe them, as these attributes are described in the list in a generic manner. For each attribute in this list, the specification provides recommendations regarding how to populate the attribute.

- Correlation Ids - They are supplied to correlate two notifications that have been generated because of a related cause. This attribute is optional. But in case of alarms that are generated to clear certain conditions, i.e., produced with a perceived severity of *SA_NTF_SEVERITY_CLEARED*, the correlation id shall be populated by the application with the notification Id that was generated by the Notification Service while invoking the *saNtfNotificationSend()* API during the production of the actual alarm.
- Event Time - The application might pass a timestamp or optionally pass an *SA_TIME_UNKNOWN* value in which case the timestamp is provided by the Notification Service.
- NCI Id - The *vendorId* portion of the *SaNtfClassIdT* data structure must be set to *SA_NTF_VENDOR_ID_SAF* always. The *majorId* and *minorId* will vary based on the specific SA Forum service and the particular notification. Every SA Forum

service shall have a *majorId* as described in the enumeration *SaNtfSafServicesT* of the Notification Service specification. The *minorIds* will be described and reused on a per-service basis.

- Notification Id - This attribute is obtained from the Notification Service when a notification is generated, and hence need not be populated by an application.
- Notifying Object - DN of the entity generating the notification. This name must conform to the SA Forum AIS naming convention and 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. For details on the AIS naming convention, refer to the Overview document.

9.2 Availability Management Framework Notifications

The following sections describe a set of notifications that an Availability Management Framework implementation shall produce.

The value of the *majorId* field within the notification Class identifier (*SaNtfClassIdT)* should be set to as follows in all notifications generated by the Availability Management Framework.

- *majorId* = SA_SVC_AMF

The *minorId* field within the notification class identifier (*SaNtfClassIdT)* is set distinctly for each individual notification as described below. This field is range-bound, and the used ranges are:

- Alarms: (0x01 - 0x64)
- State change notifications: (0x65 - 0xC8)
- Object change notifications: (0xC9 - 0x12C)
- Attribute change notifications: (0x12D - 0x190)

9.2.1 Availability Management Framework Alarms

9.2.1.1 Availability Management Framework Service Impaired

Description

The Availability Management Framework is currently unable to provide service or is in a degraded state because of certain issues with memory, resources, communication, or other constraints.

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 Availability Management Framework self-healed or recovered and is again providing service.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Event Type	Mandatory	*SA_NTF_ALARM_COMMUNICATION*
Notification Object	Mandatory	AMF Service, same as Notifying object as specified above.
Notification Class Identifier	NTF internal	*minorId* = 0x01
Additional Text	Optional	“AMF service impaired.”
Additional Information ID	Optional	Unspecified
Probable Cause	Mandatory	Applicable value from enum *SaNtfProbableCauseT* in [2]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum *SaNtfSeverityT* in [2]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

9.2.1.2 Component Instantiation Failed

Description

The Availability Management Framework was unable to successfully instantiate a particular component. This means that the INSTANTIATION command invoked on the component either returned an error exit status or failed to successfully complete within the time period specified by the INSTANTIATE timeout, and all subsequent attempts by the Availability Management Framework to revive the component, includ-

ing a possible node reboot did not resolve the issue causing the component to enter the instantiation-failed presence state. For more details, refer to Section 4.4.

Clearing Method

Manual, after taking the appropriate administrative action

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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* = 0x02
Additional Text	Optional	“Instantiation of Component <LDAP DN of component> failed”
Additional Information ID	SA Forum 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 [2]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum *SaNtfSeverityT* in [2]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

9.2.1.3 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 4.6.

Clearing Method

Manual, after taking the appropriate administrative action

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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* = 0x03
Additional Text	Optional	“Cleanup of Component <LDAP DN of component> failed”
Additional Information ID	SA Forum 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 [2]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum *SaNtfSeverityT* in [2]
Trend Indication	Optional	Unspecified

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

9.2.1.4 Cluster Reset Triggered by a Component Failure

Description

A component failed and recommended to the Availability Management Framework a cluster reset recovery action, i.e., *SA_AMF_CLUSTER_RESET*.

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.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Event Type	Mandatory	*SA_NTF_ALARM_PROCESSING*
Notification Object	Mandatory	LDAP DN of the component, which recommended an *SA_AMF_CLUSTER_RESET* recovery
Notification Class Identifier	NTF internal	*minorId* = 0x04
Additional Text	Optional	“Failure of Component <LDAP DN of component> triggered cluster reset.”
Additional Information ID	Optional	Unspecified

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Probable Cause	Mandatory	Applicable value from enum *SaNtfProbableCauseT* in [2]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum *SaNtfSeverityT* in [2]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

9.2.1.5 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 3.3.3.2.

This alarm is typically generated when the Availability Management Framework is unable to successfully execute a recovery in case of a failure (node/service unit, etc.) in order to prevent the service disruption and maintain service availability. This alarm should be also generated when an administrative action renders a service instance unassigned.

Clearing Method

Manual, after taking the appropriate administrative action

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Event Type	Mandatory	*SA_NTF_ALARM_PROCESSING*
Notification Object	Mandatory	LDAP DN of the service instance, which has no current active assignments.
Notification Class Identifier	NTF internal	*minorId* = 0x05
Additional Text	Optional	“SI designated by <LDAP DN of the SI> has no current active assignments to any SU.”
Additional Information ID	Optional	Unspecified
Probable Cause	Mandatory	Applicable value from enum *SaNtfProbableCauseT* in [2]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum *SaNtfSeverityT* in [2]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

9.2.1.6 Proxied component becomes unproxied

Description

This alarm is generated by the Availability Management Framework when it reliably confirms the fact that a component that was being proxied previously is no longer proxied, i.e., the Availability Management Framework has not been able to engage another component to assume the mediation responsibility for a component whose designated proxy component has failed.

Clearing Method

Manual, after taking the appropriate administrative action.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Event Type	Mandatory	*SA_NTF_ALARM_PROCESSING*
Notification Object	Mandatory	LDAP DN of component, which is no longer proxied.
Notification Class Identifier	NTF internal	*minorId* = 0x06
Additional Text	Optional	Unspecified
Additional Information ID	Optional	Unspecified
Probable Cause	Mandatory	Applicable value from enum *SaNtfProbableCauseT* in [2]
Specific Problems	Optional	Unspecified
Perceived Severity	Mandatory	Applicable value from enum *SaNtfSeverityT* in [2]
Trend Indication	Optional	Unspecified
Threshold Information	Optional	Unspecified
Monitored Attributes	Optional	Unspecified
Proposed Repair Actions	Optional	Unspecified

9.2.2 Availability Management Framework State Change Notifications

9.2.2.1 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.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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	*minorId* = 0x65 for Node *minorid* = 0x66 for SU *minorid* = 0x67 for SG *minorid* = 0x68 for SI *minorid* = 0x69 for Application *minorid* = 0x6A for Cluster.
Additional Text	Optional	Unspecified
Additional Information ID	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*

9.2.2.2 Operational State Change Notify

Description

The operational state of a node or a service unit changed.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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* = 0x6B for Node *minorid* = 0x6C for SU
Additional Text	Optional	Unspecified
Additional Information ID	Optional	Unspecified
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*

9.2.2.3 Presence State Change Notify

Description

The presence state change of a service unit is reported only if it becomes instantiated, uninstantiated, or restarting.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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* = 0x6D
Additional Text	Optional	Unspecified
Additional Information ID	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*

9.2.2.4 HA State Change Notify

Description

The HA state of a service unit changes for an assigned service instance.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended Value
Event Type	Mandatory	*SA_NTF_OBJECT_STATE_CHANGE*
Notification Object	Mandatory	LDAP DN of the service unit whose HA state changed on behalf of a particular SI.
Notification Class Identifier	NTF internal	*minorId* = 0x6E
Additional Text	Optional	“The HA state of SI <LDAP DN> assigned to SU <LDAP DN> changed.”
Additional Information ID	SA Forum Mandatory	infoId = SA_AMF_SI_NAME, infoType = SA_NTF_VALUE_LDAP_NAME, infoValue = LDAP DN of the SI which was assigned to the SU whose HA state 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 *SaAmfPresenceStateT*

9.2.2.5 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 8.2.1.5.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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* = 0x6F
Additional Text	Optional	“The Assignment state of SI <LDAP DN of SI> changed.”
Additional Information ID	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*

9.2.2.6 Status of a component change to Proxied

Description

The status of a previously unproxied component became proxied, potentially because a proxy component assumed the task of proxying an unproxied component.

NTF Attribute Name	Attribute Type (X.73Y Recommendation or NTF)	SA Forum Recommended 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* = 0x70
Additional Text	Optional	Unspecified
Additional Information ID	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*

Appendix A Implementation of CLC Interfaces

The commands or callbacks used to control the life cycle of the various component types differ considerably. To talk conveniently about these life cycle operations, the specification uses the names instantiate, terminate and cleanup. The following table shows how the operations with that names are implemented:

Table 18 Implementation of CLC Interfaces per Component Type

Component Type	Operation	Implementation
SA-aware	instantiate	CLC-CLI INSTANTIATE
	terminate	*saAmfComponentTerminateCallback()*
	cleanup	CLC-CLI CLEANUP
proxied, pre-instantiable	instantiate	*saAmfProxiedComponentInstantiateCallback()*
	terminate	*saAmfComponentTerminateCallback()*
	cleanup	CLC-CLI CLEANUP (if local) *saAmfProxiedComponentCleanupCallback()*
proxied, non-pre-instantiable	instantiate	*saAmfCSISetCallback()*
	terminate	*saAmfCSIRemoveCallback()*
	cleanup	CLC-CLI CLEANUP (if local) *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 former is executed. Only if errors occur during that operation, is the CLEANUP command run.

Appendix B API functions in Unregistered Processes

A registered process of a component may invoke all SA API functions, and all SA API callback functions may be invoked for such processes. An unregistered process within a 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 them. For each API function of the Availability Management Framework (sorted alphabetically) and for all the remaining SA API functions in the last line as a whole, the table below indicates by a YES in the second column, whether the function can be invoked in the context of an unregistered process. A NO in the second column indicates that the function can only be invoked in the context of a registered process.

Table 19 API Functions Invoked by or on Unregistered Processes

API Interfaces	API Can be Invoked in the Context of an Unregistered Process
saAmfComponentErrorClear()	YES
saAmfComponentErrorReport()	YES
saAmfComponentNameGet()	YES
saAmfComponentRegister()	NO
SaAmfComponentTerminateCallbackT	NO
saAmfComponentUnregister()	NO
saAmfCSIQuiescingComplete(()	NO
SaAmfCSIRemoveCallbackT	NO
SaAmfCSISetCallbackT	NO
saAmfDispatch()	YES
saAmfFinalize()	YES
saAmfHAStateGet()	YES
SaAmfHealthcheckCallbackT	YES
saAmfHealthcheckConfirm()	YES

Table 19 API Functions Invoked by or on Unregistered Processes

API Interfaces	API Can be Invoked in the Context of an Unregistered Process
saAmfHealthcheckStart()	YES
saAmfHealthcheckStop()	YES
saAmfInitialize()	YES
saAmfPmStart()	YES
saAmfPmStop()	YES
SaAmfProtectionGroupTrackCallbackT	YES
saAmfProtectionGroupTrack()	YES
saAmfProtectionGroupTrackStop()	YES
SaAmfProxiedComponentCleanupCallbackT	NO
SaAmfProxiedComponentInstantiateCallbackT	NO
saAmfResponse()	YES
saAmfSelectionObjectGet()	YES
All SA services	YES

Appendix C Example for Proxy/Proxied Association

The following example outlines the procedure by which a proxied component gets associated with a proxy component and their subsequent interactions with the Availability Management Framework during the instantiation and registration phase. The example uses two SGs with different redundancy models. One containing the proxied components and the other containing the proxy components.

Proxied SGx1:

- has a 2+1 (N+M) redundancy model,
- and contains the service units SUx1, SUx2 and SUx2.
- SUx1 contains component cx1, SUx2 contains component cx2, 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,
- contains the service units SUp1 and SUp2.
- SUp1 contains component cp1, SUp2 contains component cp2.
- CSIs corresponding to the components is CSIp1 (“Proxy CSI”)
- There is only a single SI, SIx1 protected by this service group.

The AMF 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, and 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, then at that time, the AMF has the following information:

- CSIp1 is associated with the proxied components cx1, cx2, and cx3. This is from the configuration.

- CSIp1 is currently being assigned active to cp1.

So, 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 illustrates a instantiation sequence for the above sample configuration when cx1 and cx2 are instantiated and registered but cx3 does not register (potentially because of a failure).

- 1. cp1 is instantiated by AMF using INSTANTIATE command.
- 2. cp1 registers with AMF using *saAmfComponentRegister()* API.
- 3. CSIp1 is assigned active to cp1 using *SaAmfCSISetCallbackT*.
- 4. AMF invokes *SaAmfProxiedComponentInstantiateCallbackT* for cx1 on cp1.
- 5. cx1 is registered by cp1 with AMF using *saAmfComponentRegister()* API.
- 6. SA_AIS_OK is returned for step 4. to AMF by cp1 using *saAmfResponse()* API.
- 7. AMF invokes *SaAmfProxiedComponentInstantiateCallbackT* for cx2 on cp1.
- 8. cx2 is registered by cp1 with AMF using *saAmfComponentRegister()* API.
- 9. SA_AIS_OK is returned for step 7. to AMF by cp1 using *saAmfResponse()* API.
- 10. AMF invokes *SaAmfProxiedComponentInstantiateCallbackT* for cx3 on cp1.
- 11. cx3 is not registered by cp1 with AMF for some reason.(eg. failure)
- 12. Failure is returned for step 10. by cp1. (see step 15 for subsequent AMF actions)
- 13. AMF assigns CSIx1 to cx1 via cp1 using *SaAmfCSISetCallbackT*.
- 14. AMF assigns CSIx2 to cx2 via cp1 using *SaAmfCSISetCallbackT*.
- 15. AMF invokes *SaAmfProxiedComponentCleanupCallbackT* for cx3 on cp1 and carries out the regular procedure to try to revive cx3 which if fails transitions cx3 to INSTANTIATION-FAILED presence state raises the alarm etc.

Note: in the scenario described above, CSIx3 was never assigned, which would have been the case for an SA-aware component also.