# Service Availability™ Forum
# Service Availability Interface

Overview SAI-Overview-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

# 1 Document Introduction

## 1.1 Document Purpose

This document (**SAI-Overview-B.02.01**) provides a brief guide to the remainder of the Service Availability™ Forum (SA Forum) Interface Specifications documents and includes a description of the SA Forum Information Model.

As the Application Interface Specification (AIS) APIs are described in several documents, this document also provides an introduction to the AIS documents. It describes the objectives of the AIS specification as well as programming models and definitions that are common to all specifications. Additionally, it contains an overview of the Availability Management Framework and of the other AIS services and a chapter that describes the main abbreviations, concepts and terms used in the AIS documents.

This revision of the document does not contain a complete overview of the access interfaces (SNMP) and the hardware interface specification (HPI), this is intended to be provided in future versions of the document.

## 1.2 Documents Organization

The **SAI-XMI-A.01.01** document contains the description of SA Forum Information Model in XML Metadata Interchange (XMI) v1.2 format.

### 1.2.1 Hardware Interface Specification (HPI) Documents

The Hardware Interface Specification is organized into the following documents:

- **SAI-HPI-B.01.01** describes the HPI API.
- **SAIM-HPI-B.01.01-ATCA** describes the mapping on ATCA platforms.
- **SAI-HPI-SNMP-B.01.01** describes the HPI SNMP MIBs.

### 1.2.2 Application Interface Specification (AIS) Documents

The Application Interface Specification is organized into the following documents:

- **SAI-AIS-AMF-B.02.01** describes the Availability Management Framework API.
- **SAI-AIS-CLM-B.02.01** describes the Cluster Membership Service API.
- **SAI-AIS-CKPT-B.02.01** describes the Checkpoint Service API.
- **SAI-AIS-EVT-B.02.01** describes the Event Service API.
- **SAI-AIS-MSG-B.02.01** describes the Message Service API.
- **SAI-AIS-LCK-B.02.01** describes the Lock Service API.

- **SAI-AIS-IMM-A.01.01** describes the Information Model Management Service API.
- **SAI-AIS-LOG-A.01.01** describes the Log Service API.
- **SAI-AIS-NTF-A.01.01** describes the Notification Service API.
- **SAI-AIS-SNMP-A.01.01** the AIS SNMP MIBs.

## 1.3 History

This section presents the main changes in the current release of this document (SAI-Overview-B.02.01), with respect to its previous release (SAI-AIS-B.01.01). The editorial changes are not mentioned.

### 1.3.1 New Topics

- Section 2.2 on page 13 includes the description of the new AIS Services: Information Model Management Service (IMMS), Notification Service (NTF), and Log Service (LOG).
- Section 2.4 on page 18 introduces the new AIS SNMP MIBs.
- Section 3.1.5.4 on page 36 explains the service-specific functions used to deallocate memory allocated by the area service library when tracking an object.
- Section 3.3.3 on page 43 introduces new floating point types, Section 3.3.4 on page 43 introduces the new *SaStringT* type, and Section 3.3.8 on page 45 describes the new *SaAnyT* type.
- Section 3.3.16 on page 53 introduces the new SA_AIS_ERR_NO_OP and SA_AIS_ERR_REPAIR_PENDING error codes.
- Section 3.3.9.1.2 on page 48 shows the values for the *safApp* RDN of AIS services.
- Chapter 4 on page 57 provides a UML description of the SA Forum Information Model and its mapping on the Information Model Management Service.

### 1.3.2 Clarifications

- Section 3.1.1.1 on page 25 on asynchronous APIs explains that the choice is left to the implementation whether errors are detected in the library and returned by the asynchronous API or whether errors are detected by the area server and returned subsequently by the callback.
- Section 3.1.2.1 on page 27 clarifies the consequences of a process exit after the process has successfully called the area server initialization function and before it invokes the corresponding area server finalization function.
- Section 3.3.9.1 on page 47 clarifies that only printable Unicode characters can be used in LDAP names.

### 1.3.3 Changes

- Section 3.1.4.2 on page 32 states now that each AIS service must provide functions to free memory dynamically allocated by AIS service functions.
- Section 3.1.5.1 on page 34 states now that It is the responsibility of the calling process to invoke the corresponding free function of the area service library to deallocate the buffer allocated by the area service library if the *items* fields in the *Sa<Area><Object>NotificationBufferT* is NULL.
- The format of Distinguished Names (DNs) of the different SA Forum objects are shown in Chapter 4 on page 57. The RDN for an external service unit has been removed and *safSu* is valid now for any service unit and not only for local service units. Some new RDNs have been added.
- Section 3.3.16 on page 53: The description of the SA_AIS_ERR_FAILED_OPERATION error code has been generalized and the SA_AIS_ERR_NAME_NOT_FOUND has been removed, as it was not used in the AIS specifications.

## 1.4 References

The following documents contain information that is relevant to the specification:

[1] CCITT Recommendation X.730 | ISO/IEC 10164-1, Object Management Function

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

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

[4] CCITT Recommendation X.736 | ISO/IEC 10164-7, Security Alarm Reporting Function

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

[6] http://www.unicode.org

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 of the Application Interface Specification

This chapter provides an overview of the Application Interface Specification (AIS), which includes the AIS Availability Management Framework and the AIS Services. More details on the AIS Availability Management Framework and on the AIS Services, and the APIs that they provide, can be found in corresponding documents.

This chapter also includes a section describing the dependencies among the AIS Services and a section introducing the AIS SNMP MIBs.

## 2.1 AIS Availability Management Framework

The Availability Management 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 cluster that consists of a number of cluster nodes. These nodes host various resources in a distributed computing environment.

The Availability Management Framework provides a set of APIs to enable highly available applications. The Availability Management Framework determines the states of a component and monitors the health of components. It also allows a component to query the Availability Management Framework for information about the component's state.

## 2.2 AIS Services

The following core AIS Services provide the basic functionality of the cluster on which the Availability Management Framework and the highly available application are implemented. The AIS Services defined in this specification are:

- Cluster Membership Service
- Checkpoint Service
- Event Service
- Message Service
- Lock Service
- Information Model Management Service
- Notification Service
- Log Service

Each of these services is briefly described below.

### 2.2.1 Cluster Membership Service

The Cluster Membership Service provides applications with membership information about the nodes that have been administratively configured in the cluster configuration (these nodes are also called cluster nodes or configured nodes), and it is core to any clustered system. A cluster consists of this set of configured nodes, each with a unique node name.

A member node is a configured node that the Cluster Membership Service has recognized to be healthy and well-connected to be used for deploying HA applications and services. The set of member nodes at a given point in time is referred to as the cluster membership, or simply membership. The Cluster Membership Service is the authority that decides whether a configured node is transitioned to be a member node of the cluster.

The Cluster Membership Service also allows application processes to register a callback function to receive membership change notifications as those changes occur.

### 2.2.2 Checkpoint Service

The Checkpoint Service provides a facility for processes to record checkpoint data incrementally, which can be used to protect an application against failures. When processes recover from a failure (with a restart or a fail-over procedure), the Checkpoint Service can be used to retrieve the previous checkpoint data and resume execution from the state recorded before the failure, minimizing the impact of the failure.

Checkpoints are cluster-wide entities. A copy of the data stored in a checkpoint is called a checkpoint replica, which is typically stored in main memory rather than on disk for performance reasons. A given checkpoint may have several checkpoint replicas stored on different nodes in the cluster to protect it against node failures.

### 2.2.3 Event Service

The Event Service is a publish/subscribe multipoint-to-multipoint communication mechanism that is based on the concept of event channels: One or more publishers communicate asynchronously with one or more subscribers via events over a cluster-wide entity, named event channel. Publishers can also be subscribers on the same event channel.

Events consist of a standard header and zero or more bytes of published event data. The Event Service API does not impose a specific layout for the published event data.

### 2.2.4 Message Service

The Message Service specifies a buffered message passing system based on the concept of a message queue for processes on the same or on different nodes. Mes-

sages are written to and read from message queues. A single message queue permits a multipoint-to-point communication. Message queues are persistent or non-persistent. The Message Service must preserve messages that have not yet been consumed when the message queue is closed.

Processes sending messages to a message queue are unaware that the process, which was originally processing these messages, has been replaced by another process acting as a standby in case the original process fails or switches over.

Message queues can be grouped together to form message queue groups. Message queue groups permit multipoint-to-multipoint communication. They are identified by logical names, so that a process is unaware of the number of message queues and of the physical location of the message queues to which it is communicating. The sender addresses message queue groups using the same mechanisms that it uses to address single message queues. The message queue groups can be used to distribute messages among message queues pertaining to the message queue group. Regardless of the number of message queues to which messages are distributed, the message queue group remains accessible under the same name.

Message queue groups can be used to maintain transparency of the sender process to faults in the receiver processes, represented by the message queues in the message queue groups. The sender process communicates with the message queue group. If a receiver process fails, the sender process continues to communicate with the message queue group and is unaware of the fault, because it continues to obtain service from the other receiver processes.

With message queues, the Message Service uses the model of *n* senders to *one* receiver whereas, with message queue groups, the Message Service uses the model of *m* senders to *n* receivers.

### 2.2.5 Lock Service

The Lock Service is a distributed lock service, intended for use in a cluster, where processes in different nodes might compete with each other for access to a shared resource.

The Lock Service provides entities, called lock resources, that are used to synchronize access to shared resources between application processes.

The Lock Service provides a simple lock model supporting two locking modes for exclusive access and shared access.

### 2.2.6 Information Model Management Service

The different entities of an SA Forum cluster, such as components provided by the Availability Management Framework, checkpoints provided by the Checkpoint Ser-

vice, or message queues provided by the Message Service are represented by various objects of the SA Forum information model.

The SA Forum information model (IM) is specified in UML and managed by the Information Model Management (IMM) Service.

The objects in the Information Model are provided with their attributes and administrative operations (i.e., operations that can be performed on the represented entities through system management interfaces). For management applications or Object Managers, the IMM provides the APIs to create, access and manage these objects.

Subsequently, it delivers the requested operations to the appropriate AIS services or applications (referred to as Object Implementers) that implement these objects for execution.

Information Model objects and attributes can be classified into two categories:

- Configuration objects and attributes
- Runtime objects and attributes

The IMM Service exposes two sets of APIs:

(1) An Object Management API (OM-API) exposed typically to system management applications (for example, SNMP agents and CIM providers).

(2) An Object Implementer API (OI-API) restricted to Object Implementers.

### 2.2.7 Notification Service

The Notification Service is, to a great degree, based on the ITU-T Fault Management model as found in the X.700 series of documents as well as many other supportive recommendations.

The Notification Service is centered around the concept of a notification, which explains an incident or change in status. The term ‘notification’ is used instead of ‘event’ to clearly distinguish it from ‘event’ as defined by the AIS Event Service.

There are five notification types with distinct parameters. They are:

- Alarm
- State Change
- Object Creation/Deletion
- Attribute Value Change
- Security Alarm

The Notification Service is based on a publish/subscribe paradigm.

Any number of notification producers can publish notifications.

Notification consumers can be of two types:

- Notification subscribers receive selected notifications as they occur.
- Notification readers retrieve historical notification entries from the persistent notification log.

There can be any number of notification consumers of each type.

### 2.2.8 Log Service

SA Forum distinguishes between a Log and a Trace Service: The former is for cluster-significant, function-based information suited for system administrators or automated tools, while the later is low level implementation-specific information suited for developers or field engineers.

The Log Service enables applications to express and forward log records through well-known log streams that lead to particular output destinations such as a named file. Once at the output destination, a log record is subject to configurable and public output formatting rules. Since the output format is public, third party tools can read these log files.

There are four types of log streams supported by the log service:

- the alarm log stream for ITU X.733 [3] and ITU X.736 [4] based log records,
- the notification stream for ITU X.730 [1] and ITU X.731 [2] based log records,
- the system stream is for system relevant log records, and
- the application stream is for application-specific log records.

For each of the alarm, notification, and system log stream types, there is exactly one log stream in an SA Forum cluster. However, there can be any number of application log streams, each with a unique name, that can come and go, as needed by running applications.

### 2.2.9 Modeling AIS Services

The SA Forum AIS does not specify any particular implementation of the various AIS services that are described in Section 2.2. However, the SA Forum strongly recommends using the system modeling abstractions and logical entities that are made available by the Availability Management Framework specification while implementing such services. This promotes a single and unified AIS modeling scheme (based on Availability Management Framework logical entities) and causes the AIS services to be modeled, managed, and upgraded in the same way as any other SA Forum application would be modeled, managed, and upgraded.

## 2.3 Dependencies

AIS services including the Availability Management Framework have a dependency on the AIS Notification Service. The Notification Service in turn has a dependency on the Log Service.

The Availability Management Framework and all AIS services, except the Information Model Management Service and the Notification Service, have a section that describes the various alarms and notifications that may be generated by these services. These alarms and notifications are expressed using the notification producer API syntax and semantics as specified in the Notification Service. The expectation is that an alarm correlator, an element manager, or a management subagent in the cluster would subscribe for notifications and alarms (using the notification subscriber API) that they are interested in.

The Information Model Management Service exposes the configuration objects and the administrative operations on behalf of all AIS services. Therefore, any service that exposes such a management interface depends on the IMM Service.

Other Interactions between the AIS Availability Management Framework and the AIS Services, and between the AIS Services, may depend on functions that are not defined by this version of the specification and may be defined by future versions of the specification.

## 2.4 SNMP MIBs

In order to provide system management access to various entities exposed by the Availability Management Framework and the various other AIS services, SNMP MIBS are defined for these services.

SNMP (Simple Network Management Protocol) is an application layer protocol that facilitates the exchange of management information between networked devices or systems to provide for management access, monitoring, and control. The SNMP model assumes the existence of managers and agents.

The SNMP agents implementing the AIS MIBs rely on the Information Model management (IMM) Service to access the various managed objects exposed by AIS Services.

# 3 AIS Programming Model and Naming Conventions

This chapter describes the programming model and naming conventions used by the SA Forum Application Interface Specification (AIS). The chapter aims to ensure uniformity in the specifications of the various AIS services.

The chapter discusses the asynchronous and synchronous APIs, and the APIs for using a library of the Application Interface Specification. The chapter describes how the names of type, data, function and macro declarations are formed, and defines the predefined types and constants, which support application portability between platforms and implementations.

## 3.1 Programming Model and Usage Overview

This section provides an overview of the SA Forum Application Interface programming model and the generally intended usage of the SA Forum Application Interfaces. The descriptions contained herein are not intended to constrain implementations unduly.

The SA Forum Application Interface occurs between a process and a library that implements the interface. The interface is designed for use by both threaded and non-threaded application processes.

The term **process**, as used in this document, can be regarded as being equivalent to a process defined by the POSIX standard. However, the use of the term process does not mandate a POSIX process but, rather, any equivalent entity that a system provides to manage executing software.

The **area server** is an abstraction that represents the server that provides services for a specification area (Availability Management Framework, Cluster Membership Service, Checkpoint Service, Event Service, Message Service and Lock Service). Each area has a separate logical area server, although the implementer is free to create them as the same or separate physical modules.

The area implementation libraries may be implemented in one or several physical libraries; however, a process is required to initialize, register and obtain an operating system "selection object" separately for each area's implementation library. Thus, from a programming standpoint, it is useful to view these as separate libraries.

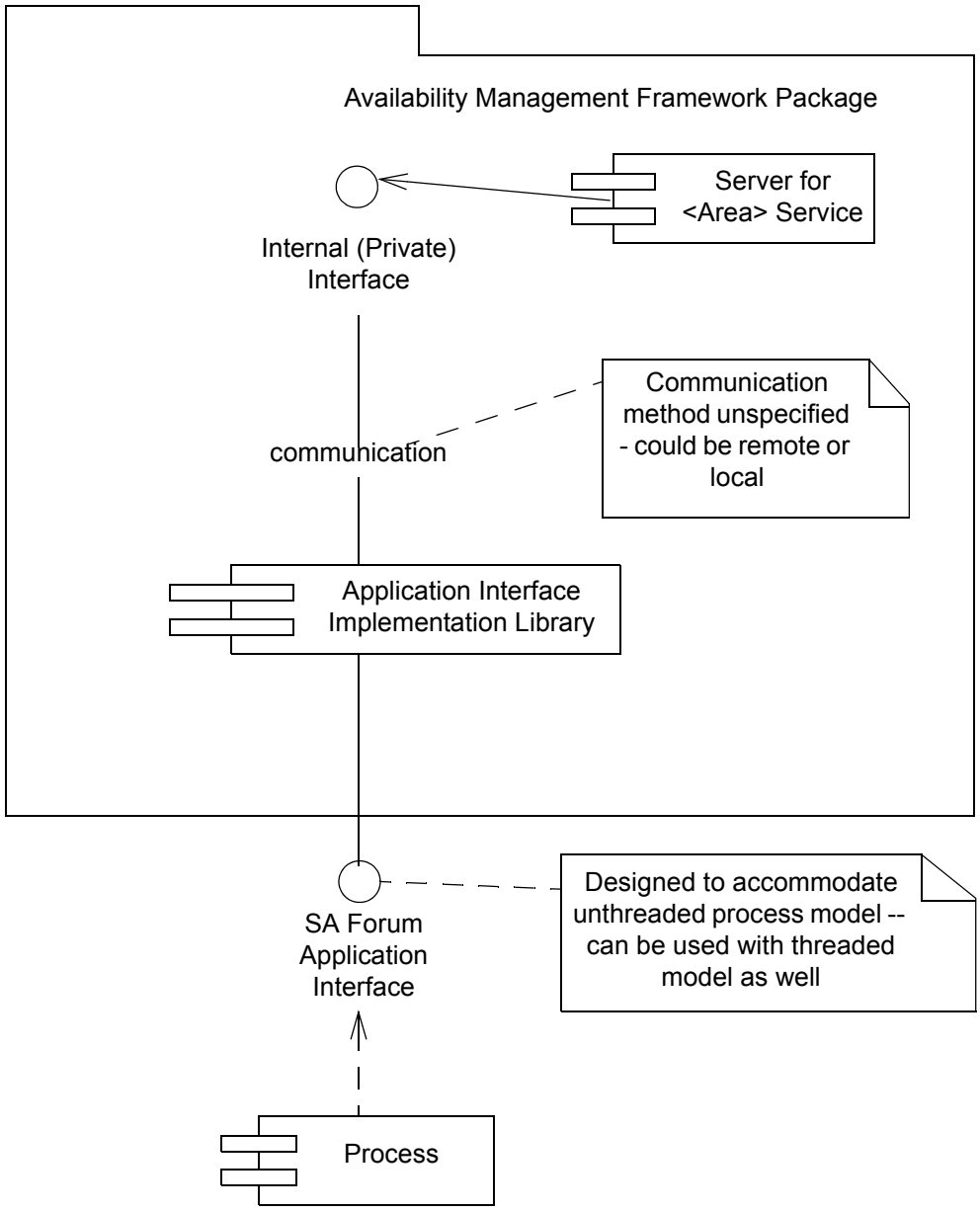
The UML diagram in Figure 1 shows the relationships among an "area" server, an “area” implementation library, and a process, all represented as UML components.

Although Figure 1 shows only one area server, area implementation library, and application component, there is nothing restricting an area server from interfacing with numerous area implementation libraries and an area implementation library from servicing multiple application components. Where a component comprises multiple processes, each process must have its own implementation libraries and must initialize those libraries itself.

Note: For those readers who are unfamiliar with UML, the boxes with two rectangles on the left are UML "components" (not to be confused with components in the context of the SA Forum Application Interface Specification), the box with a "tab" at the top is a package, and the two circles are interfaces. The dashed lines to the interfaces are dependency or "consumes" relationships, and the solid lines to the interfaces are "realizes" or "provides" relationships. Thus, the process connected to the interface by the dashed line is an interface consumer, while that connected by the solid line is an interface provider. As shown in Figure 1, the area server and the area implementation library are packaged together.

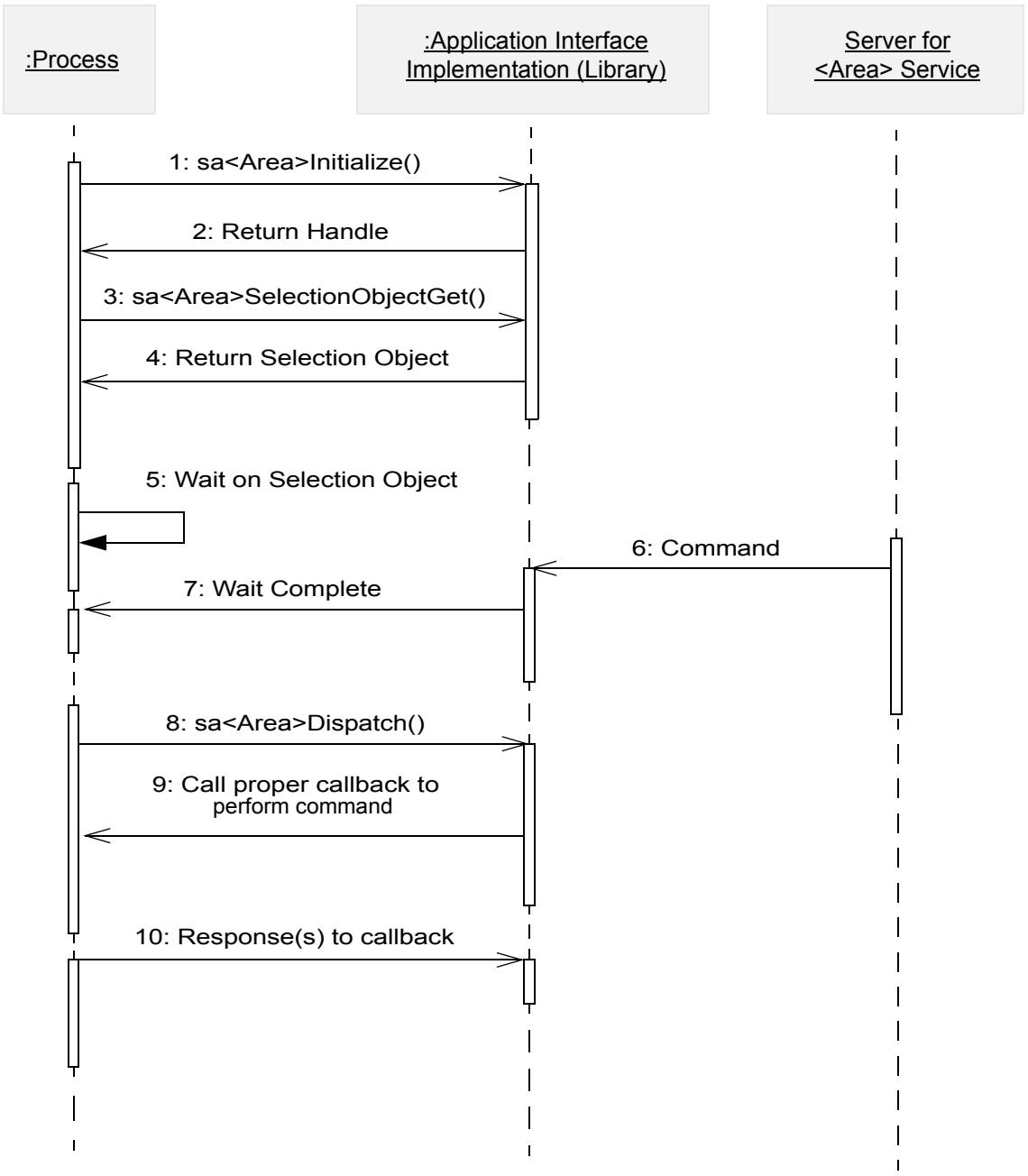
It is expected that the area server and area implementation library are packaged together and are designed to be released as a set. However, this does not preclude providing other packaging options.
The interface between the area server and the area implementation library is proprietary and outside the scope of this specification. The area server and the area implementation library could reside on the same or separate computers, and perhaps even within the same software module.

**FIGURE 1 Interface Relationships**



The SA Forum Application Interface Specification programming/usage model views the area server as a "server" for the component and the component as a "client" of the area server. In this sense, the usage model is not much different from that of an X-windows "client" application, where setup is done and the application then receives callbacks as events occur.

**FIGURE 2 Programming/Usage Model**




The programming/usage model is shown in Figure 2. Again, this model is intended to show the usage for which the interfaces were generally intended, rather than unduly constraining implementations. For example, it is possible that actions 6 and 7 of the

model might be combined, or the library might obtain the command from the area server between actions 8 and 9.

An example usage of the APIs, which involves a callback mechanism, is as follows:

1. The process within the component invokes the *sa<Area>Initialize()* function to provide a set of callbacks for use by the library in calling back the component.
2. The *sa<Area>Initialize*() function returns an interface handle to the invoking process.
3. The process invokes the *sa<Area>SelectionObjectGet()* function to obtain a selection object, which is an operating system dependent object (e.g., a file descriptor suitable for use in *select()* for Unix/Linux).
4. The interface returns a selection object to the process. This operating system dependent object allows the process to wait until an invocation to a callback function is pending for it.
5. The process waits on the selection object.
6. The area server sends a command over its "private" interface to the library.
7. The library "awakes" the selection object, thereby awaking the process.
8. The process invokes the *sa<Area>Dispatch()* function.
9. The library invokes the appropriate callback function of the process corresponding to the command received from the area server. The callback function parameters inform the process of the specific details of the command issued by the area server or the information provided by the area server.
10. Once the process completes processing the callback, it responds by invoking a function of the area interface. In some cases, more than one response invocation, or no response, may be necessary.

In addition to the callback mechanism, certain functions that the component may invoke are asynchronous, for example, for obtaining information from the area server, via the library, or for reporting errors.

### 3.1.1 Synchronous and Asynchronous Programming Models

The Application Interface Specification employs both the synchronous and asynchronous programming models. The synchronous model is generally easier for programmers to understand and use. However, there are situations where the number of simultaneous outstanding requests precludes having an independent thread of execution for each. Some applications also require direct control of scheduling within a process. To support such applications, asynchronous APIs are used in the core of the

service availability components.
AIS defines synchronous and asynchronous variants of open calls, as it is expected that these are cluster-operations needing some time to complete. In contrast, only synchronous close calls are specified, as it is expected that these calls return as soon as possible to the caller, and that the remaining processing is done asynchronously.

Synchronous APIs are generally used for library and association housekeeping interfaces. It is possible to build a synchronous API on top of an asynchronous API but not vice versa.

Note: Some of the examples in this section contain POSIX operating system specific constructs. The examples are given for illustrative purposes only and do not imply that POSIX specific constructs are necessary to use a given programming model.

#### *3.1.1.1 Asynchronous APIs*

Functions that are called by an application process and that solicit an asynchronous response from the area server, e.g., those with an Async suffix, generally have as the first two parameters *<area>Handle* and *invocation*. The *<area>Handle* is the handle that was provided by the sa<Area> library when the process invoked the *sa<Area>Initialize()* function. This allows the sa<Area> library to invoke the response callback function, using the correct selection object in a multithreaded process.

The process allocates *invocation* for the call, and uses *invocation* subsequently to distinguish the corresponding response invocation. Response invocations that are solicited by a process have *invocation* as the first parameter. Unsolicited callbacks, such as asynchronous event notifications, do not require *invocation*.

If the API implementation does not invoke the callback function, for whatever reason, then the process receives no other indication of the completion or success of the asynchronous function that it invoked.

Typically, the choice is left to the implementation whether errors are detected in the library and returned by the asynchronous API or whether errors are detected by the area server and returned subsequently by the callback. In order to allow this flexibility, some error codes are listed as returned values of the asynchronous API as well as errors returned by the callback. If an error is detected directly by the asynchronous API (which typically means that the return value from the API is different from SA_AIS_OK), the request for the corresponding asynchronous operation is implicitly canceled and no callback is invoked subsequently for this operation.

## Example

An asynchronous function declaration:

```
SaAisErrorT saClmClusterNodeGetAsync(
        SaClmHandleT clmHandle,
        SaInvocationT invocation,
        SaClmClusterNodeIdT nodeId,
        SaClmClusterNodeT *clusterNode
);
```

The corresponding response declaration:

```
typedef void (*SaClmClusterNodeGetCallbackT)(
        SaInvocationT invocation,
        const SaClusterNodeInfoT *clusterNode,
        SaAisErrorT error
);
```

#### *3.1.1.2 Synchronous APIs*

Two types of synchronous APIs do not need any particular consideration:

1. A synchronous API that does not require a context switch, i.e., it can be completed by local processing within the library.
2. A synchronous API that will not, or may not, be called from a function with bounded time constraints.

Other APIs and, in particular, the synchronous counterparts of asynchronous APIs provide a timeout parameter to control the blocking behavior of the call.

### Example

```
SaAisErrorT error;
SaClmClusterNodeT clusterNode;
SaClmNodeIdT nodeId;
SaTimeT timeout; /* timeout value for synchronous invocations */
...
timeout = 100 * SA_TIME_ONE_MILLISECOND; /* 100 milliseconds */
nodeId = 10;
error = saClmClusterNodeGet(clmHandle, nodeId, timeout, &clusterNode);
if (error == SA_AIS_ERR_TRY_AGAIN) { /* try again later */ }
```

## 3.1.2 Library Life Cycle

#### *3.1.2.1 Initialization*

The use of a Service Availability library starts with a call to initialize the library. This call potentially loads any dynamic code and binds the asynchronous calls implemented by the process.

### Prototype

```
SaAisErrorT sa<Area>Initialize(
        Sa<Area>HandleT *<area>Handle,
        const Sa<Area>CallbacksT *<area>Callbacks,
        const SaVersionT *version
);
```

A handle *<area>Handle,* representing the association of the library initialization, is returned by the library and used in subsequent asynchronous calls and for finalization. Service availability libraries must support several invocations of *sa<Area>Initialize()* issued from the same binary program (e.g., process in the POSIX.1 world). Each invocation to *sa<Area>Initialize()* returns a different handle. This allows a process to obtain a separate *selectionObject* for each handle, thereby allowing support for multithreaded dispatching of *<area>* callbacks.

When a process invokes an asynchronous function of the *<area>* library, the *<area>Handle,* cited as a parameter of that function, can determine the selection object that the library uses for the asynchronous response callback.

The *<area>Callbacks* parameter contains a structure of pointers to the functions implemented by the process that the *<area>* library can invoke. If the process does not implement any callback functions, it must invoke *sa<Area>Initialize()* with a NULL *<area>Callbacks* parameter. The process must also set individual members of

*Sa<Area>CallbacksT* to NULL, if these particular callbacks are not to be used by the particular initialization and must not be invoked by the *<area>* library.

Any API calls, including *sa<Area>Dispatch()* calls (refer to Section 3.1.2.3 on page 29), can be called from any callback function.

If the invoking process exits after successfully returning from the *sa<Area>Initialize()* function and before invoking *sa<Area>Finalize()* to finalize the handle *<area>Handle* (see Section 3.1.2.2 on page 28), the *<Area> Service* automatically finalizes this handle and any other handles obtained via the handle *<area>Handle* when the death of the process is detected.

## Prototype

*typedef void (*SaComponent<Object><Action>T)(...);*

## Example

```
typedef void (*SaClmClusterNodeGetCallbackT)(
        SaInvocationT invocation,
        SaClmClusterNodeT *clusterNode,
        SaAisErrorT error,
);
```

The process calls structure prototype:

```
typedef struct {
Sa<Area><Object><Action-1>CallbackT sa<Area><Object><Action-1>Callback;
Sa<Area><Object><Action-2>CallbackT sa<Area><Object><Action-2>Callback;
...
Sa<Area><Object><Action-N>CallbackT sa<Area><Object><Action-N>Callback;
} Sa<Area>CallbacksT;
```

As an input parameter of the *sa<Area>Initialize()* function, *version* indicates the version of the Availability Management Framework or the particular SA service that the process requires. This parameter can be used by library implementers to provide support for different API versions in a single library. As an output parameter, the version actually supported by the Availability Management Framework or the particular SA service is delivered.

#### *3.1.2.2 Finalization*

When the process no longer requires the use of the area functions, it calls the area finalization function. The semantics of finalization are area-dependent, concerning termination of outstanding requests; however, the intention is to disassociate the pro-

cess from the interface area implementation instance and recover any associated resources. If a process has invoked *sa<Area>Initialize()* multiple times to obtain multiple *<area>Handles,* it must invoke *sa<Area>Finalize()* separately for each such handle.

## Prototype

*SaAisErrorT sa<Area>Finalize(Sa<Area>HandleT <area>handle);*

where the handle parameter is the handle returned by the corresponding prior invocation of the initialization function.

#### *3.1.2.3 Dispatching*

In the synchronous model, the dispatching of Service Availability interface area library calls is done when the process makes a call to an API of the area. This interaction may depend on some IPC or synchronization primitives that might be blocking. If synchronous versions of the APIs are used in a non-threaded environment, polling by repeatedly invoking the call with a small timeout value might be required to service multiple requests simultaneously.

Dispatching in the asynchronous model is supported by obtaining an operating system handle that allows the process to ascertain whether there are any calls pending. The generic call to obtain the operating system handle is as follows:

```
SaAisErrorT sa<Area>SelectionObjectGet(
        Sa<Area>HandleT <area>Handle,
        SaSelectionObjectT *selectionObject
);
```

In the POSIX.1 world, the selection object is simply a file descriptor, provided by the operating system, and *selectionObject* is a pointer to the file descriptor. The *selectionObject* returned by *sa<Area>SelectionObjectGet()* is valid until *sa<Area>Finalize()* is invoked on *<area>Handle*.

The following code fragment illustrates how to detect pending area invocations for various library associations referenced by the handle parameter of the corresponding *sa<Area>SelectionObjectGet()* calls. Note there may be multiple active handles for the same area.

## Example

```
#define MAX_AREA 5

SaSelectionObjectT fd[MAX_AREA];
void (*dispatch[MAX_AREA])();
SaUint32T *handle[MAX_AREA];
SaUint32T handle0;
SaUint32T handle1;
...
int i;
fd_set rfds;
int nfds = 0;
int numArea = 0;
struct timeval timeout;
sa<Area0>SelectionObjectGet(handle0, &fd[numArea]);
dispatch[numArea] = (void *) sa<Area0>Dispatch;
handle[numArea] = &handle0;
numArea++;
sa<Area1>SelectionObjectGet(handle1, &fd[numArea]);
dispatch[numArea] = (void *) sa<Area1>Dispatch;
handle[numArea] = &handle1;
numArea++;
...
FD_ZERO(&rfds);
for (i=0; i<numArea; i++) {
        if (nfds < fd[i]) nfds = fd[i]; /* find max fd */
        FD_SET(fd[i], &rfds);
}
select(nfds+1, &rfds, NULL, NULL, &timeout);
for (i=0; i<numArea; i++) {
        if (FD_ISSET(fd[i], &rfds)) (*dispatch[i])(*handle[i], SA_DISPATCH_ONE);
}
```

When the process detects that invocations are pending for a library association and is ready to process them, it calls the relevant *sa<Area>Dispatch()* function. This invocation may be made in the main thread or in a dedicated thread. Dispatching with different priorities can be achieved by initializing multiple associations each with a dedicated thread running at the appropriate operating system priority.

### Prototype

*SaAisErrorT sa<Area>Dispatch(*

*Sa<Area>HandleT <area>Handle,*
*SaDispatchFlagsT dispatchFlags*

*);*

The *<area>Handle* is obtained from the *sa<Area>Initialize()* function, and the *dispatchFlags* specify the callback execution behavior of the *sa<Area>Dispatch()* function. The *sa<Area>Dispatch()* function invokes, in the context of the calling thread, pending callbacks for the handle, designated by *<area>Handle,* in a way that is specified by the *dispatchFlags* parameter.

If no callbacks are pending and *sa<Area>Dispatch()* is invoked with either the SA_DISPATCH_ONE or the SA_DISPATCH_ALL flags, it returns immediately and successfully. Refer to Section 3.3.13 on page 51 for the meaning of the SA_DISPATCH_ONE and SA_DISPATCH_ALL flags.

Different threads of a process can invoke *sa<Area>Dispatch()* on the same handle. As a consequence, several pending callbacks may be invoked concurrently. It is up to the application to provide concurrency control (for instance, locking), if needed.

## 3.1.3 Interaction Between AIS and POSIX APIs

In a POSIX environment, the AIS functions can be invoked concurrently by different threads of a process. Hence, the AIS functions must be thread-safe. However, this specification does not require that the AIS functions can be safely invoked from a signal handler.

When developed in a POSIX environment, greater portability of applications from one AIS implementation to another can be attained by observing the following rules during application development:

- Avoid using any SA Forum API from a signal handler.
- Do not assume that SA Forum APIs are interruptible by signals.
- Do not assume that SA Forum APIs are thread cancellation points.
- Do not assume that the AIS functions are fork-safe. Therefore, if a process using AIS functions forks a child process, in which AIS functions will be called, the child process should exec() a new program immediately after being forked. This new program can, then, use AIS functions.

## 3.1.4 Memory Management

### *3.1.4.1 Usage of [in], [out], and [in/out] in Parameters*

The AIS services use the acronyms [in], [out], and [in/out] in the description of parameters. These acronyms have the following meaning:

- [in] is used when a parameter passes information to the invoked function and receives no information from the invoked function.
- [out] is used when the caller passes a memory area through a pointer, and no additional information for the invoked function is passed in this memory area. The invoked function supplies the requested information into the provided memory area.
- [in/out] is used when a parameter passes information to the invoked function and receives information from the invoked function.

### *3.1.4.2 Memory Allocation and Deallocation*

**Rule 1**

Memory dynamically allocated by one entity (user process or service area library) is deallocated by the same entity that allocated it. This rule has only one exception, described in rule 2 below.

**Rule 2**

In the following cases, it is simpler to have the area service library allocate the buffer and have the service user deallocate the memory:

- It is not easy to provide a buffer of the appropriate size by the invoking process as it is hard to predict in advance how much memory is actually required.
- Avoid excessive copying for performance reasons.

This use must be clearly documented, because it is a potential source of memory leaks.

Each area service providing a function that dynamically allocates memory for a user process must provide a function to be called by the user to deallocate the allocated memory.

The following prototype definitions and a code sample illustrate the use of rule 2.

## Prototype

```
typedef struct{
        char *buf;
        SaInt32T len;
} SaXxxBufferT;

SaAisErrorT saXxxReceive(SaXxxHandleT handle, SaXxxBufferT buffer);

SaAisErrorT saXxxReceiveDataFree(SaXxxHandleT handle, void *buffer);
```

## Example

```
SaXxxxBufferT msg;
SaInt32T myLen;
msg.buf = NULL;
error = saXxxReceive(handle, msg);
if (error != SA_AIS_OK) { /* handle error */ }
if (msg.buf != NULL) {
        /* process message */
        myLen = msg.len; /* area service sets length */
        process_message(msg.buf, myLen);
        saXxxReceiveDataFree(handle, msg.buf);
        msg.buf = NULL;
};
```

#### 3.1.4.3 Handling Pointers in a Process and an Area Service

The following notes explain how a service user process and the area service should handle pointers passed as parameters:

- When the area service library invokes a callback function, provided by the process, and that callback function has a parameter that is a pointer, the process must not retain that pointer after the callback function has returned. Rather, if the process needs to retain that information, it must copy the information into memory that it has allocated.

- When the process invokes a synchronous function, provided by the area service, the area service must not retain any pointer, passed to it as a parameter of that function, after the function has returned.
- When the process invokes an asynchronous function, provided by the area service, the area service must not retain any pointer, passed to it as a parameter of that function, after it has invoked the corresponding asynchronous callback function.

### 3.1.5 Track APIs

Some area services provide groups of entities and allow these groups to be tracked by service user processes. For example, the Message Service allows tracking the membership of message queues within message queue groups.

The track APIs of services providing them, are not identical, but very similar. They consist of three functions:

- Track an object
- Stop tracking an object
- Callback notification about an object change

The format of a function name is:

*sa<Area><Object>Track[<Func>]()*

where *<Area>, <Object>,* and *<Func>* denote the area service, the tracked object, and one of the track functions, respectively.

A tracked object is identified by an area service handle and an object name. The object name is omitted if the object can be identified by the area service handle only. For this reason, the *objectName* parameter in the APIs listed below is marked as "if needed".

Examples:

- A tracked queue group is identified by the Message Service handle and the queue group name.
- A tracked cluster membership is identified by the Cluster Membership Service handle. It has no separate object name.

#### *3.1.5.1 Track an Object*

A call to the routine

*SaAisErrorT sa<Area><Object>Track(*

*Sa<Area>HandleT <area>Handle,*
*SaNameT *objectName, /* if needed */*
*SaUin8T trackFlags,*
*Sa<Area><Object>NotificationBufferT *notificationBuffer*

);

tracks the object in a way determined by the *trackFlags* parameter (see Section 3.3.12 on page 51).

If the flag SA_TRACK_CURRENT is set in the *trackFlags* parameter, initial status information of the tracked object is retrieved. If the *notificationBuffer* parameter is not NULL, this information is passed in the given buffer; otherwise, it is passed asynchronously via the callback notification API.

The *notificationBuffer* is of type:

*typedef struct{*

*/* Optional fields specific to the service */*
*SaUint32T numberOfItems;*
*Sa<Area><Object>NotificationT *items;*

*} Sa<Area><Object>NotificationBufferT;*

If *items* is NULL, the area service will allocate the buffer. The required information will be placed by the service library into the allocated buffer when the *sa<Area><Object>Track()* call returns. It is the responsibility of the calling process to invoke the corresponding free function of the area service library to deallocate the allocated buffer (see Section 3.1.5.4 on page 36).

Status changes of the tracked object are always passed asynchronously through the callback notification API; however, if the *trackFlags* parameter contains no flag other than SA_TRACK_CURRENT, a one-time status request is made. No subsequent status changes are notified, unless they have been requested in a preceding *sa<Area><Object>Track()* call.

#### *3.1.5.2 Callback Notification*

If a process has called *sa<Area><Object>Track()* such that asynchronous notifications will take place, these are passed through the callback

*typedef void (*Sa<Area><Object>TrackCallbackT)(*

*SaNameT *objectName, /* if needed */*
*Sa<Area><Object>NotificationBufferT *notificationBuffer,*
*SaUint32T numberOfMembers,*
*SaErrorT error*

);

The *notificationBuffer* contains the information of the tracked object according to the *trackFlags* parameter in a preceding *sa<Area><Object>Track()* call. It is always allocated by the area service, and it cannot be accessed outside the callback routine.

The *numberOfMembers* parameter contains the number of members in the group represented by the tracked object.

#### *3.1.5.3 Stop Tracking an Object*

A call to the routine

*SaAisErrorT sa<Area><Object>TrackStop(*

*Sa<Area>HandleT <area>Handle,*
*SaNameT *objectName /* if needed */*

);

stops tracking an object. No more callback notifications about object status changes will be sent to the process.

This call is only needed if there was a preceding invocation to *sa<Area><Object>Track()*, and if this invocation was not a one-time status request for the object.

#### *3.1.5.4 Deallocating Memory Allocated for Tracking an Object*

A call to the routine

*SaAisErrorT sa<Area><Object>NotificationFree(*

*Sa<Area>HandleT <area>Handle,*
*Sa<Area><Object>NotificationT *items*

);

deallocates the memory, pointed to by the *items* parameter. This memory was allocated by the area service library in a previous call to the *sa<Area><Object>Track()* function.
For details when this memory is allocated, refer to the description of the *items* field in the *Sa<Area><Object>NotificationBufferT* structure (see Section 3.1.5.1 on page 34)

## 3.2 Naming Conventions

The conventions for the naming of constants, types, variables and functions defined in the SA Forum Application Interface Specification are covered in this section. The Application Interface Specification is broken down into interface areas. An interface area consists of a set of self-contained APIs that can be provided as a single library with its associated header file(s). Each interface area is assigned an interface area tag (or simply area tag, if the context makes it clear) that identifies the functions pertaining to a given area.

Application interface area tags:

- *Hpi* ::= Hardware Platform Interface
- *Amf* ::= Availability Management Framework
- *Clm* ::= Cluster Membership Service
- *Ckpt* ::= Checkpoint Service
- *Evt* ::= Event Service
- *Msg* ::= Message Service
- *Lck* ::= Lock Service
- *Imm* ::= Information Model Management Service
- *Ntf* ::= Notification Service
- *Log* ::= Log Service

<*Area*> used in names (see next sections) consists of the interface area tag followed by an optional Sub-area tag:

<*Area*> = <*Area tag*> [<*Sub-area tag*>]

The <*Sub-area tag*> is currently only defined for the Information Model Management Service. Two values are defined for the <*Sub-area tag*> of this service:

- Om for Object Management
- Oi for Object Implementer

### 3.2.1 Case Sensitivity

All usage of strings in the AIS documents is assumed to be case sensitive, and an implementation of the Availability Management Framework or the AIS services must not make any assumptions regarding the strings being case insensitive, especially for processing and comparison purposes.

### 3.2.2 Global Function Declarations

The function name of a global declaration, that is, one that is visible to an application component, has a prefix that starts with the letters “sa” in lower case, standing for “service availability”, followed by <Area> that identifies the area of the specification. The rest of the function name is formed from capitalized words that are descriptive of the object, action and tag of the function.

#### Prototype

*type sa<Area><Object><Action><Tag>(<arguments>);*

where *sa* = prefix for Service Availability

- *<Area>* = interface area
- *<Object>* = name or abbreviation of object or service
- *<Action>* = name or abbreviation of action
- *<Tag>* = tag for the function such as Async or Callback

#### Example without <Sub-area Tag>

```
SaAisErrorT saEvtChannelOpen(
        const SaEvtHandleT evtHandle,
        const SaNameT *channelName,
        SaEvtChannelOpenFlagsT channelOpenFlags,
        SaTimeT timeout,
        SaEvtChannelHandleT *channelHandle
);
```

Here, *<Area>* = Evt for Event Service, *<Object>* = Channel, and *<Action>* = Open.

#### Example with <Sub-area Tag>

```
SaAisErrorT saImmOmCcbObjectDelete(
        SaImmCcbHandleT ccbHandle,
        const SaNameT *objectName
);
```

Here, <*Area*> = ImmOm for the Object Management sub-area of the Information Model Management Service, <*Object*> = Object, and <*Action*> = Delete.

Some other common <Action> suffixes are:

- Request
- Response
- Set
- Get

For functions that solicit an asynchronous invocation from the area service, the prototype has an Async suffix unless it is obvious from the <*Action*> suffix. The corresponding callback invocation function prototype has a Callback suffix.

## Example

```
SaAisErrorT saClmClusterNodeGetAsync(
        SaClmHandleT clmHandle,
        SaInvocationT invocation,
        SaClmNodeIdT nodeId,
        SaClmClusterNodeT *clusterNode
);
```

Here, <*Area*> = Clm for Cluster Membership Service, <*Object*> = ClusterNode, <*Action*> = Get, and <*Tag*> = Async.

```
typedef void (*SaClmClusterNodeGetCallbackT)(
        SaInvocationT invocation,
        const SaClmClusterNodeT *clusterNode,
        SaAisErrorT error
);
```

Here, <*Area*> = Clm for Cluster Membership Service, <*Object*> = ClusterNode, <*Action*> = Get, and <*Tag*> = Callback.

### 3.2.3 Global Variable Declarations

The name of a global variable, that is, one that is visible to an application component, has a prefix that starts with the letters “sa”, standing for service availability, followed by <Area> that identifies the area of the specification. The rest of the name is formed from capitalized words that describe the variable.

#### Prototype

*<type> sa<Area><Variable Name>*

#### Example

*SaNameT saAmfComponentName;*

### 3.2.4 Type Declarations

The names of types, that are visible to an application component, have a prefix that starts with the letters “Sa”, followed by <Area> that identifies the area of the specification. The rest of the name is formed from capitalized words that describe the type.

#### Prototype

*typedef <...> Sa<Area><TypeName>T*

#### Example

*typedef SaUint32T SaCkptHandleT;*

*typedef SaUint32T SaEvtChannelOpenFlagsT;*

### 3.2.5 Macro Declarations

The names of macros that are visible to an application component use only uppercase letters and the digits 0-9. Underscores are used to separate words and improve readability. Macro names start with the letters “SA”, followed by an underscore, followed by <Area> followed by an underscore and underscore separated words.

#### Prototype

*#define SA_<AREA>_<MACRO NAME> <macro definition>*

#### Example

*#define SA_EVT_HIGHEST_PRIORITY 0*

### 3.2.6 Enumeration Type Declarations

The names of enumeration elements use only uppercase letters and the digits 0-9. Underscores are used to separate words and improve readability. Element names

start with the letters “SA”, followed by an underscore, followed by <Area>, followed by an underscore and underscore separated words.

## Prototype

*typedef enum {*

*SA_<AREA>_<ENUMERATION_NAME1> [= <value>],*
*SA_<AREA>_<ENUMERATION_NAME2> [= <value>],*
*....*
*SA_<AREA>_<ENUMERATION_NAMEn> [= <value>]*

*} <enumeration type name>;*

## Example

*typedef enum {*

*SA_CKPT_SECTION_VALID = 1,*
*SA_CKPT_SECTION_CORRUPTED = 2*

*} SaCkptSectionStateT;*

## 3.3 Standard Predefined Types and Constants

### 3.3.1 Boolean Type

The type *SaBoolT* defines the standard boolean type.

#### Definition

```
typedef enum {
        SA_FALSE = 0,
        SA_TRUE = 1
} SaBoolT;
```

### 3.3.2 Signed and Unsigned Integer Types

The set of fixed bit-width integer types expected to be supported on all platforms are defined below.

#### *3.3.2.1 Signed Types*

- Signed 8 bit integer quantity: *SaInt8T*
- Signed 16 bit integer quantity: *SaInt16T*
- Signed 32 bit integer quantity: *SaInt32T*
- Signed 64 bit integer quantity: *SaInt64T*

A typical declaration of these types on a 32-bit platform is as follows:

```
typedef char SaInt8T;

typedef short SaInt16T;

typedef long SaInt32T;

typedef long long SaInt64T;
```

#### *3.3.2.2 Unsigned Types*

- Unsigned 8-bit integer quantity: *SaUint8T*
- Unsigned 16 bit integer quantity: *SaUint16T*
- Unsigned 32 bit integer quantity: *SaUint32T*
- Unsigned 64 bit integer quantity: *SaUint64T*

A typical declaration of these types on a 32-bit platform is as follows:

*typedef unsigned char SaUint8T;*
*typedef unsigned short SaUint16T;*
*typedef unsigned long SaUint32T;*
*typedef unsigned long long SaUint64T;*

### 3.3.3 Floating Point Types

Two floating point types are defined:

*typedef float SaFloatT;*
*typedef double SaDoubleT;*

### 3.3.4 String Type

#### Definition

*typedef char * SaStringT;*

#### Example

*typedef SaStringT SaImmClassNameT;*

### 3.3.5 Size Type

This *SaSizeT* type is used to specify sizes of objects.

#### Definition

*typedef SaUint64T SaSizeT;*

#### Example

*SaSizeT checkpointSize;*

### 3.3.6 Offset Type

This *SaOffsetT* type is used to specify offsets in data areas.

### Definition

*typedef SaUint64T SaOffsetT;*

### Example

*SaOffsetT dataOffset;*

## 3.3.7 Time Type

The *SaTimeT* type is used to specify time values. A time value is always expressed as a positive number of nanoseconds, except for the SA_TIME_UNKNOWN constant, defined later in this section.

The *SaTimeT* type can be interpreted as either an absolute timestamp or a time duration.

An interface specification containing a parameter of *SaTimeT* type should state how the time value is interpreted. If no such statement is present, a duration value is assumed.

### Definition

*typedef SaInt64T SaTimeT;*

### Granularity

Nanoseconds = $10^{-9}$ seconds

### Range

Approximately 292 years

In some cases, it is necessary to represent an unknown time value. A special value is reserved for this:

### Definition

*#define SA_TIME_UNKNOWN 0x8000000000000000*

This hexadecimal constant corresponds to a time of $-2^{63}$ nanoseconds.

#### *3.3.7.1 Timestamps*

A timestamp is represented in an *SaTimeT* data item as the number of positive nanoseconds elapsed since 00:00:00 UTC, 1 Jan 1970.

It is common to use another term called “absolute time” that is the same as the definition of a timestamp. These two terms are often used interchangeably.

## Definition

*#define SA_TIME_END 0x7FFFFFFFFFFFFFFF*

SA_TIME_END represents the largest timestamp value: Fri Apr 11 23:47:16.854775807 UTC 2262.

## Definition

*#define SA_TIME_BEGIN 0x0*

SA_TIME_BEGIN represents the smallest timestamp value: Thu 1 Jan 00:00:00 UTC 1970.

#### *3.3.7.2 Time Durations*

A time duration is represented in an *SaTimeT* data item as the number of positive nanoseconds counted from a given reference time, for instance, the time of an API call.

For the convenience of the user, the following values are defined:

| | |
|---|---|
| *#define SA_TIME_ONE_MICROSECOND* | *1000* |
| *#define SA_TIME_ONE_MILLISECOND* | *1000000* |
| *#define SA_TIME_ONE_SECOND* | *1000000000* |
| *#define SA_TIME_ONE_MINUTE* | *60000000000* |
| *#define SA_TIME_ONE_HOUR* | *3600000000000* |
| *#define SA_TIME_ONE_DAY* | *86400000000000* |
| *#define SA_TIME_MAX* | *SA_TIME_END* |

A duration of SA_TIME_MAX is interpreted as an infinite duration. If a timeout parameter is set to SA_TIME_MAX when invoking an AIS API function, there will be no time limit associated with this request. This value should be viewed as a convenience value for programmers who do not care about timeouts associated with various APIs. Typically, it is not advisable to use SA_TIME_MAX in timeout parameters, and other pre-defined constants should generally suffice.

### 3.3.8 Sequence of Octets Type

This *SaAnyT* type is used to define a set of arbitrary octets.

## Definition

*typedef struct {*

*SaSizeT buffer Size;*
*SaUint8T *bufferAddr;*

*} SaAnyT;*

## Example

*SaAnyT SaRawBinary;*

### 3.3.9 Name Type

The type *SaNameT* is intended to be used to represent object names that are passed in SA Forum APIs. It allows for both human readable and other representations. Human readable representations include ASCII and multi-byte character locales. The length field in the *SaNameT* structure refers to the number of octets (bytes) used by the representation of the name in the name field. If the C character string representation is used, the value field contains the characters in the string without the terminating null character, and the length field the number of these characters.

## Definition

```
#define SA_MAX_NAME_LENGTH 256

typedef struct {
        SaUint16T length;
        SaUint8T value[SA_MAX_NAME_LENGTH];
} SaNameT;
```

## Example

```
SaNameT myName;
...
myName.length = strlen("fred");
memcpy(myName.value,"fred", myName.length);
error = saXxxCreateObject(myName,yyy,zzz);
```

#### *3.3.9.1 Note on AIS Object Names*

Current AIS runtime APIs use LDAP distinguished names (DNs) to name objects. Future AIS runtime and system management interfaces (XML, administrative APIs, configuration management APIs) will also use LDAP DNs to name objects. These LDAP DNs follow UTF-8 encoding conventions described in reference [5].

The scope of these names is limited to the cluster. Hence, the names do not include any relative distinguished name (RDN), which would identify the cluster itself.

Multi-valued RDNs are not supported.

LDAP names are encoded in *SaNameT* by using its UTF-8 representation without a terminating null character. Only printable Unicode characters must be used in LDAP names. This simplifies printing or displaying these names (see [6]).

Section 4.1 on page 58 indicates for each object class the supported formats for the DNs of their object instances.

##### 3.3.9.1.1 Recommendations on RDN Values

- The value of a node RDN should be identical to the operating system node name, if this notion is supported by the operating system running on the node.
- To minimize name conflicts, RDN values for components, message queues, message queue groups, checkpoints, event channels or locks should include a

prefix specific to the particular application they are associated with. For example the stock symbol of the company providing the application is an example for such prefix.

- When exposed through the AIS interfaces, these DNs are encapsulated in an *SaNameT* data structure and normalized as follows:
  - All tabs or white spaces before or after '=' separating the RDN type from the RDN value, and before or after the ',' character separating the RDNs, are removed.
  - Only ',' is used to separate RDNs.
  - Because *SaNameT* has a size of 256 characters, the size of the RDN values represented as UTF-8 strings is limited to 64 characters.

#### 3.3.9.1.2 Values for the *safApp* Application RDN of AIS Services

This section describes standard SA Forum AIS defined RDN values for the *safApp* RDN for the various AIS services. The values use a common format of *safApp* = *saf*<*Area*>*Service*[:<*varAppName*>] where the *saf*<*Area*>*Service* part has constant well-known values as defined below for the various AIS services and the <*varAppName*> is an arbitrary string (according to rules defined in Section 3.3.9.1).

- Availability Management Framework — *safApp*="safAmfService"
- Checkpoint Service — *safApp*="safCkptService"
- Cluster Membership Service — *safApp*="safClmService"
- Event Service — *safApp*="safEvtService"
- Information Model Management Service — *safApp*="safImmService"
- Lock Service — *safApp*="safLckService"
- Message Service — *safApp*="safMsgService"
- Notification Service — *safApp*="safNtfService"
- Log Service — *safApp*="safLogService"

The <*varAppName*> part of the RDN value enables differentiation of multiple implementations of the same AIS service.

### 3.3.10 SaServicesT

The following type enumerates the services specified by the SA Forum.

```
typedef enum {
        SA_SVC_HPI = 1,
        SA_SVC_AMF = 2,
        SA_SVC_CLM = 3,
        SA_SVC_CKPT = 4,
        SA_SVC_EVT = 5,
        SA_SVC_MSG = 6,
        SA_SVC_LCK = 7,
        SA_SVC_IMMS = 8,
        SA_SVC_LOG = 9,
        SA_SVC_NTF = 10
} SaServicesT;
```

### 3.3.11 Version Type

The *SaVersionT* type is used to represent software versions of area implementations. Application components can use instances of this type to request compatibility with a particular version of an SA Forum Application Interface area specification. The area referred to is implicit in this API.

## Definition

```
typedef struct {
        SaUint8T releaseCode;
        SaUint8T majorVersion;
        SaUint8T minorVersion;
} SaVersionT;
```

*releaseCode*: The release code is a single ASCII capital letter [A-Z]. All specifications and implementations with the same release code are backward compatible. Refer to Section 3.3.11.1 for details on how the SA Forum will handle backward compatibility. It is expected that the release code will change very infrequently. Release codes are assigned exclusively by the SA Forum.

*majorVersion*: The major version is a number in the range [01..255]. An area implementation with a given major version number implies compliance to the interface specification bearing the same release code and major version number. Changes to a specification requiring a revision of the major version number are expected to occur at most once or twice a year for the first few years, becoming less frequent as time goes on. Major version numbers are assigned exclusively by the SA Forum.

*minorVersion:* The minor version is a number in the range [01..255]. Successive updates to an area implementation complying to an area interface specification bearing the same release code and major version number have increasing minor version number starting from 01. Increasing minor version numbers only refer to enhancements of the implementation, like better performance or bug fixes. Different values of the minor version may not impact the compatibility and are not used to check whether required and supported versions are compatible.

Successive updates to an area interface specification with the same release code and major version number will also have increasing minor version numbers starting from 01. However, such changes to a specification are limited to editorial changes that do not impose changes on any software implementations for the sake of compliance. Minor version numbers are assigned independently by the SA Forum for interface specifications and by members and licensed implementers for their implementations.

## Example

```
SaVersionT myAmfVersion;
...
myAmfVersion.releaseCode = 'B';
myAmfVersion.majorVersion = 0x02;
myAmfVersion.minorVersion= 0x00;
/* Version "B.02.xx" */
error = saAmfInitialize(handle, const &callbacks, *myAmfVersion);
```

#### *3.3.11.1 Notes on Backward Compatibility*

To achieve backward compatibility with B.01.01 when evolving the B spec in the future, the SA Forum will follow the rules below. However, this goal can only be achieved with the cooperation of AIS implementers (refer to the notes below).

- A function/type definition never changes for a given SA Forum release.
- Changes in a function/type definition (adding a new argument to a function, adding a new field to a data structure) force the definition of a new function/type name. A new function/type name is built from the original name in B.01.01 with a suffix indicating the version where the function/type changed (for instance, *saAmfComponentRegister_3()*).
- As an exception to the previous rule, new enum values, flag values or union fields can be added to an existing enum, flag or union typedef without changing

the type name as long as the size of the enum, flag or union typedef does not change.

- AIS implementers must ensure that they respect the version numbers provided by the application when it initializes the library and do not expose new enum values to applications using older versions.
- AIS implementers must also ensure that they respect the version numbers provided by the application when the library is initialized, with regards to new or modified error codes and do not expose error codes that only apply to functions in the most recent version of the specification to applications written to an older version of the specification.

### 3.3.12 Track Flags

The following constants are used by the *sa<Area><Object>Track()* API for all areas with track APIs to specify what needs to be tracked and what information is supplied in the notification callback.

*#define SA_TRACK_CURRENT 0x01*

Information about all entities is returned immediately, either in a notification buffer as indicated by the caller, or by a single subsequent notification callback.

*#define SA_TRACK_CHANGES 0x02*

The notification callback is invoked each time at least one change happens in the set of entities or one attribute of at least one entity in the set changes. Information about all entities is passed to the notification callback (both entities for which a change occurred and entities for which no change occurred).

*#define SA_TRACK_CHANGES_ONLY 0x04*

The notification callback is invoked each time at least one change happens in the set of entities or one attribute of at least one entity in the set changes. Only information about entities that changed is passed in the notification callback.

### 3.3.13 Dispatch Flags

The following enumeration type is used by the dispatch function for each of the different areas.

*typedef enum {*

*SA_DISPATCH_ONE = 1,*
*SA_DISPATCH_ALL = 2,*
*SA_DISPATCH_BLOCKING = 3*

*} SaDispatchFlagsT;*

The values of the *SaDispatchFlagsT* enumeration type have the following interpretation:

- SA_DISPATCH_ONE - Invoke a single pending callback in the context of the calling thread, if there is a pending callback, and then return from the dispatch.
- SA_DISPATCH_ALL - Invoke all of the pending callbacks in the context of the calling thread, if there are any pending callbacks, before returning from dispatch.
- SA_DISPATCH_BLOCKING - One or more threads calling dispatch remain within dispatch and execute callbacks as they become pending. The thread or threads do not return from dispatch until the corresponding finalize function is executed by one thread of the process.

### 3.3.14 Selection Object

The *SaSelectionObjectT* type is used for selection objects. Selection objects are operating system dependent objects allowing a process to wait until an invocation of a callback function is pending for it.

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.

## Definition

*typedef SaUint64T SaSelectionObjectT;*

### 3.3.15 Invocation

The *SaInvocationT* type is used to match a callback call to the call requesting the callback. Refer to Section 3.1.1.1 on page 25 for details, including an example.

## Definition

*typedef SaUint64T SaInvocationT;*

### 3.3.16 Error Codes

To simplify the coding of error handling, error codes returned by SA Forum Application Interface Specification APIs are globally unique, and are defined as follows.

```
typedef enum {
        SA_AIS_OK = 1,
        SA_AIS_ERR_LIBRARY = 2,
        SA_AIS_ERR_VERSION = 3,
        SA_AIS_ERR_INIT = 4,
        SA_AIS_ERR_TIMEOUT = 5,
        SA_AIS_ERR_TRY_AGAIN = 6,
        SA_AIS_ERR_INVALID_PARAM = 7,
        SA_AIS_ERR_NO_MEMORY = 8,
        SA_AIS_ERR_BAD_HANDLE = 9,
        SA_AIS_ERR_BUSY = 10,
        SA_AIS_ERR_ACCESS = 11,
        SA_AIS_ERR_NOT_EXIST = 12,
        SA_AIS_ERR_NAME_TOO_LONG = 13,
        SA_AIS_ERR_EXIST = 14,
        SA_AIS_ERR_NO_SPACE = 15,
        SA_AIS_ERR_INTERRUPT =16,
        SA_AIS_ERR_NO_RESOURCES = 18,
        SA_AIS_ERR_NOT_SUPPORTED = 19,
        SA_AIS_ERR_BAD_OPERATION = 20,
        SA_AIS_ERR_FAILED_OPERATION = 21,
        SA_AIS_ERR_MESSAGE_ERROR = 22,
        SA_AIS_ERR_QUEUE_FULL = 23,
        SA_AIS_ERR_QUEUE_NOT_AVAILABLE = 24,
        SA_AIS_ERR_BAD_FLAGS = 25,
        SA_AIS_ERR_TOO_BIG = 26,
        SA_AIS_ERR_NO_SECTIONS = 27,
        SA_AIS_ERR_NO_OP = 28,
        SA_AIS_ERR_REPAIR_PENDING = 29
} SaAisErrorT;
```

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_VERSION - The version parameter is not compatible with the version of the implementation of the Availability Management Framework or particular service.

SA_AIS_ERR_INIT - A callback function that is required for this API has not been supplied in a previous call of *sa<Area>Initialize()*.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout or the timeout specified in the API call 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 component or process might try again later.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_BAD_HANDLE - A handle is invalid.

SA_AIS_ERR_BUSY - Resource is already in use.

SA_AIS_ERR_ACCESS - Access is denied.

SA_AIS_ERR_NOT_EXIST - An entity, referenced by the invoker, does not exist.

SA_AIS_ERR_NAME_TOO_LONG - Name exceeds maximum length.

SA_AIS_ERR_EXIST - An entity, referenced by the invoker, already exists.

SA_AIS_ERR_NO_SPACE - The buffer provided by the component or process is too small.

SA_AIS_ERR_INTERRUPT - The request was canceled by a timeout or other interrupt.

SA_AIS_ERR_NOT_SUPPORTED - The requested function is not supported.

SA_AIS_ERR_BAD_OPERATION - The requested operation is not allowed.

SA_AIS_ERR_FAILED_OPERATION - The requested operation failed.

SA_AIS_ERR_NO_RESOURCES - Not enough resources.

SA_AIS_ERR_MESSAGE_ERROR - A communication error occurred.

SA_AIS_ERR_QUEUE_FULL - The destination queue does not contain enough space for the entire message.

SA_AIS_ERR_QUEUE_NOT_AVAILABLE - The destination queue is not available.

SA_AIS_ERR_BAD_FLAGS - The flags are invalid.

SA_AIS_ERR_TOO_BIG - A value is larger than the maximum value permitted.

SA_AIS_ERR_NO_SECTIONS - There are no or no more sections matching the specified sections in the *saCkptSectionIteratorInititialize()* call.

SA_AIS_ERR_NO_OP - The requested operation had no effect.

SA_AIS_ERR_REPAIR_PENDING - The administrative operation is only partially completed as some targeted components must be repaired.

# 4 SA Forum Information Model

The SA Forum Information Model is described in UML and has been organized as fifteen UML class diagrams. Three diagrams provide global views showing how various object classes relate to each other:

- HPI View
- Cluster View
- AMF View

The other diagrams show the attributes and administrative operations of each individual class. There are six diagrams for the Availability Management Framework:

- AMF Cluster/Node Classes
- AMF Application/SG Classes
- AMF SU Class
- AMF Component Class
- AMF SI Classes
- AMF CSI Classes

And one diagram for each other AIS service:

- CKPT Classes
- CLM Classes
- EVT Classes
- LCK Classes
- MSG Classes
- LOG Classes

Section 4.1 provides the format of the Distinguished Names (DNs) of the various objects of the information model.

Section 4.2 describes how the SA Forum UML information model is implemented by the Information Model Management (IMM) Service.

The remaining sections of the chapter contain the different UML class diagrams.

Refer to SAI-XMI-A.01.01 for a UML description in XML Metadata Interchange (XMI) v1.2 format. This XMI file can be visualized with UML tools supporting XMI v1.2 (such as MagicDraw for example).

## 4.1 DN formats

**Table 1: DN formats**

| Object Class | DN format for objects of that class |
|---|---|
| *SaAmfApplication* | *"safApp=..."* |
| *SaAmfCluster* | *"safAmfCluster=..."* |
| *SaAmfComp* | *"safComp=...,safSu=...,safSg=...,safApp=..."* |
| *SaAmfCSI* | *"safCsi=...,safSi=...,safApp=..."* |
| *SaAmfCSIAssignment* | *"safCSIComp=...,safCsi=...,safSi=....,safApp=..."* |
| *SaAmfCSIAttribute* | *"safCsiAttr=...,safCsi=...,safSi=...,safApp=..."* |
| *SaAmfCSType* | *"safCSType=...,safApp=..."* |
| *SaAmfHealthcheck* | *"safHealthcheckKey=..., safComp=...,safSu=...,safSg=...,safApp=..."* |
| *SaAmfLogStreamConfig* | *"safLgStr=...,* "* |
| *SaAmfNode* | *"safAmfNode=...,safAmfCluster=..."* |
| *SaAmfSG* | *"safSg=...,safApp=..."* |
| *SaAmfSI* | *"safSi=...,safApp=..."* |
| *SaAmfSIAssignment* | *"safSISU=...,safSi=...,safApp=..."* |
| *SaAmfSIDependency* | *"safDepend=...,safSi=...,safApp=..."* |
| *SaAmfSIRankedSU* | *"safRankedSu=...,safSi=...,safApp=..."* |
| *SaAmfSU* | *"safSu=...,safSG=...,safApp=..."* |
| *SaCkptCheckpoint* | *"safCkpt=...,* "* |
| *SaCkptReplica* | *"safReplica=...,safCkpt=...,* "* |
| *SaClmCluster* | *"safCluster=..."* |
| *SaClmNode* | *"safNode=..., safCluster=..."* |
| *SaEvtChannel* | *"safChnl=...,* "* |
| *SaHpiEntity* | *"safHpiEntity=..., safHpiDomain=..."* |
| *SaLckResource* | *"safLock=...,* "* |

**Table 1: DN formats**

| Object Class | DN format for objects of that class |
|---|---|
| *SaLogStream* | *"safLgStr=..., * "* |
| *SaMsgQueue* | *"safMq=...,* "* |
| *SaMsgQueueGroup* | *"safMqg=...,* "* |
| *SaMsgQueuePriority* | *"safMqPrio=..., safMq=...,* "* |

Table 1 provides the format of the various DNs used to name objects of the SA Forum Information Model. There is one format defined for each object class. The '*' notation at the end of a DN format indicates that zero, one or more RDNs may be appended to the proposed format.

## 4.2 Mapping from UML to the IMM Service

The SA Forum Information Model is described in UML but is implemented within the Information Model Management (IMM) Service. The following is a brief description of how translation must be performed from the UML description to IMM Service concepts. Refer to SAI-AIS-IMM-A.01.01 for details about the IMM Service.

A UML class stereotype is used to indicate the IMM service class category (*CONFIG/RUNTIME*).

IMM Service attribute definition is obtained as follows:

**Table 2: Mapping attribute characteristics from UML to IMM Service**

| UML | IMM Service |
|---|---|
| *CONFIG* constraint | *SA_IMM_ATTR_CONFIG* |
| *RUNTIME* constraint | *SA_IMM_ATTR_RUNTIME* |
| *WRITABLE* constraint | *SA_IMM_ATTR_WRITABLE* |
| *CACHED* constraint | *SA_IMM_ATTR_CACHED* |
| *PERSISTENT* constraint | *SA_IMM_ATTR_PERSISTENT* |
| *RDN* constraint | *SA_IMM_ATTR_RDN* |
| [1] or [1..*] multiplicity | *SA_IMM_ATTR_INITIALIZED* |
| [0..*] or [1..*] multiplicity | *SA_IMM_ATTR_MULTI_VALUE* |

When the UML type of an attribute has no equivalent in the IMM Service, an attribute constraint is used to specify to which IMM Service type the attribute must be mapped (*SAUINT32T* or *SASTRINGT* attribute constraints).

If an attribute multiplicity is [0..1] or [0..*], the UML 'Initial Value' provides the attribute Default value.

For example, in Section 4.7 the first attribute of the *SaAmfApplication* class is described as:

*safApp: SaStringT [1] {RDN, CONFIG}*, where

- *safApp* is the attribute name,
- *SaStringT* is the attribute type,
- *[1]* is the attribute multiplicity and indicates in this case that the attribute has only one value and must be specified.
- *{RDN, CONFIG}* is the list of constraints for this attribute and indicates in this case that the attribute is the object RDN and is a configuration attribute.

As the *SaStringT* is a type supported by the IMM Service, the attribute will be implemented as an *SA_IMM_ATTR_SASTRINGT* IMM attribute.

In the same class, the second attribute is described as:

*saAmfApplicationAdminState: saAmfAdminStateT [1] {RUNTIME, CACHED, PERSISTENT, SAUINT32T},* where

- *saAmfApplicationAdminState* is the attribute name,
- *saAmfAdminStateT* is the typedef defined in the Availability Management Framework specification. The valid values for this attribute are the values of the typedef definition,
- *[1]* is the attribute multiplicity. This attribute is always present.
- *{RUNTIME, CACHED, PERSISTENT, SAUINT32T}* is the list of constraints for this attribute and indicates in this case that the attribute is a runtime attribute that is both cached by the IMM Service and persistent. *SAUINT32T* indicates that this attribute must be implemented as an *SA_IMM_ATTR_SAUINT32T* IMM attribute.

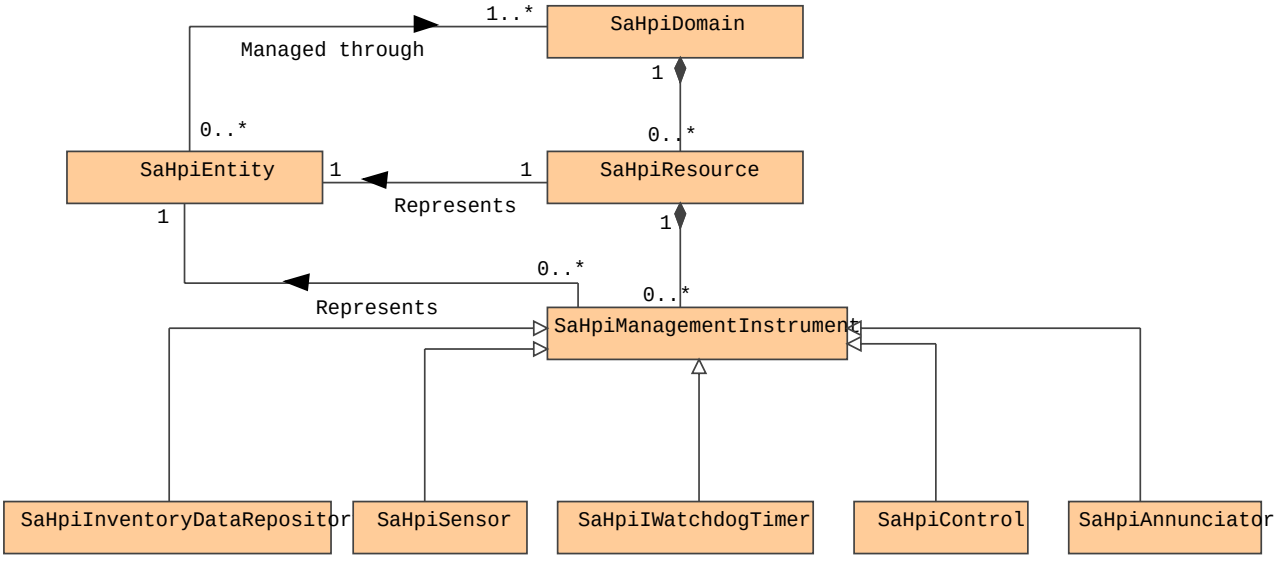
In the same section, the third attribute of the *SaAmfSG* class is described as:

*saAmfSGAutoAdjust: SaBoolT[0..1] = 0 (SA_FALSE) {CONFIG, WRITABLE,SAUINT32T},* where

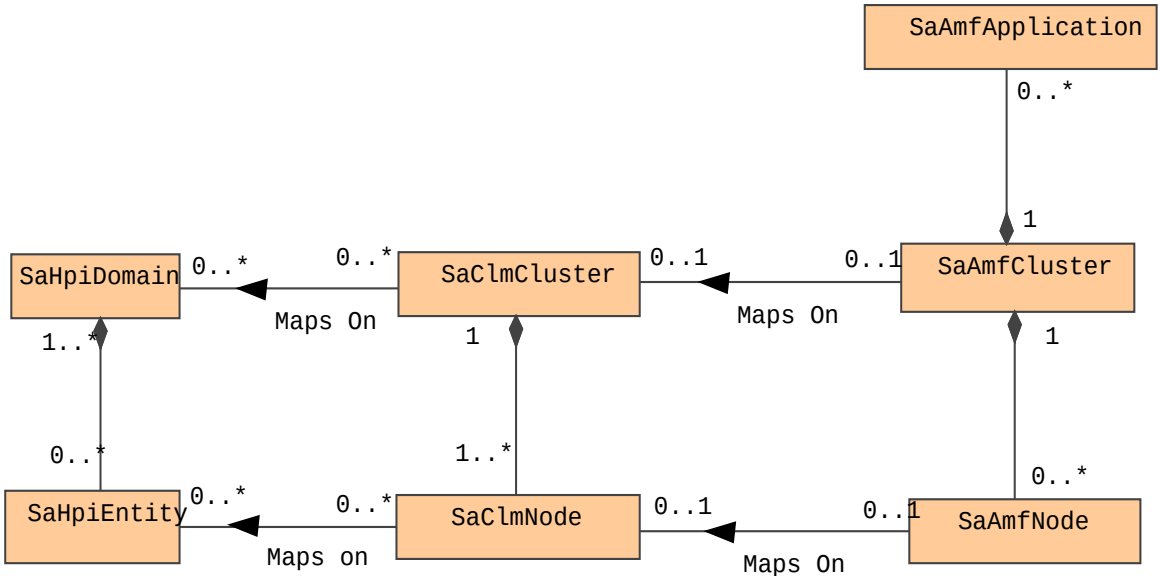
- *saAmfSGAutoAdjust* is the attribute name,

- *SaBoolT* is the typedef defined in this document on Section 3.3.1. The valid values for this attribute are the values of the typedef definition,
- *[0..1] = 0 (SA_FALSE)* indicates that this attribute can only take a single value but that the value is optional. If the value is not defined for this attribute, a default value of *0* must be taken.
- *{CONFIG, WRITABLE,SAUINT32T}* is the list of constraints for this attribute and indicates in this case that the attribute is a configuration attribute and that its value can be updated dynamically after the object has been created. *SAUINT32T* indicates that this attribute must be implemented as an *SA_IMM_ATTR_SAUINT32T* IMM attribute.

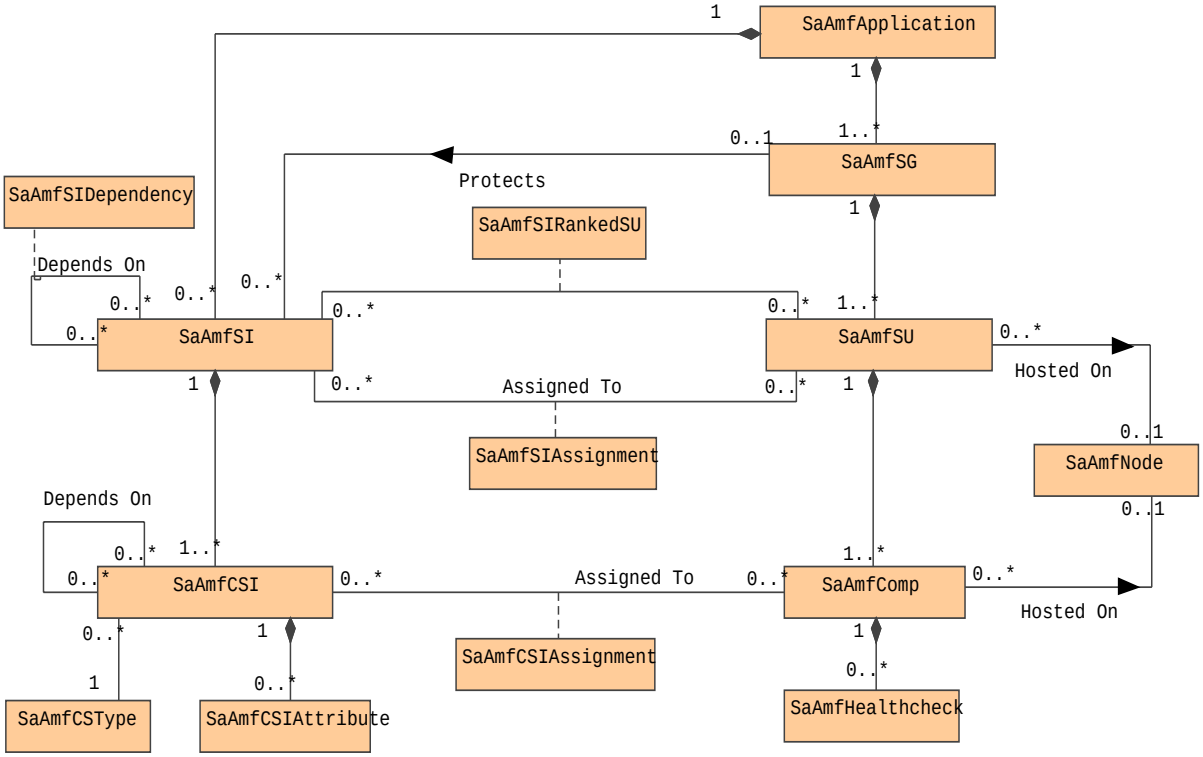
## 4.3 HPI View


SaHpiDomain
1..*
Managed through
1
0..*
SaHpiEntity
1
1
SaHpiResource
Represents
1
0..*
1
0..*
Represents
SaHpiManagementInstrument
SaHpiInventoryDataRepositor
SaHpiSensor
SaHpiIWatchdogTimer
SaHpiControl
SaHpiAnnunciator

## 4.4 Cluster View



## 4.5 AMF View


SaAmfApplication
SaAmfSG
SaAmfSI
SaAmfSU
SaAmfSIDependency
SaAmfSIRankedSU
SaAmfSIAssignment
SaAmfNode
SaAmfCSI
SaAmfComp
SaAmfCSType
SaAmfCSIAttribute
SaAmfCSIAssignment
SaAmfHealthcheck
Protects
Depends On
Assigned To
Hosted On
Depends On
Assigned To
Hosted On

## 4.6 AMF Cluster and Node Classes

| <<CONFIG>><br>SaAmfCluster |
|---|
| safAmfCluster : SaStringT [1]{ RDN, CONFIG}<br>saAmfClusterClmCluster : SaNameT [0..1] = Empty{CONFIG}<br>saAmfClusterStartupTimeout : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfClusterAdminState : SaAmfAdminStateT [1]{RUNTIME, CACHED, PERSISTENT, SAUINT32T} |
| SA_AMF_ADMIN_LOCK()<br>SA_AMF_ADMIN_SHUTDOWN()<br>SA_AMF_ADMIN_UNLOCK()<br>SA_AMF_ADMIN_LOCK_INSTANTIATION()<br>SA_AMF_ADMIN_UNLOCK_INSTANTIATION()<br>SA_AMF_ADMIN_RESTART() |

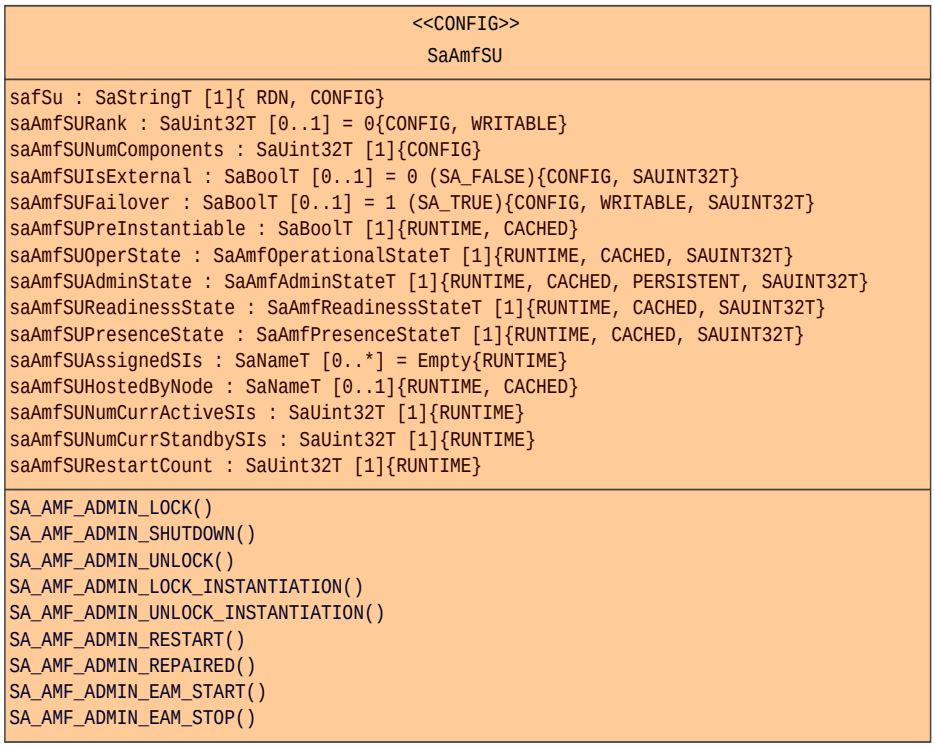
| <<CONFIG>><br>SaAmfNode |
|---|
| safAmfNode : SaStringT [1]{ RDN, CONFIG}<br>saAmfNodeClmNode : SaNameT [0..1] = Empty{CONFIG, WRITABLE}<br>saAmfNodeSuFailOverProb : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfNodeSuFailoverMax : SaUint32T [1]{CONFIG, WRITABLE}<br>saAmfNodeAutoRepair : SaBoolT [1] = 1 (SA_TRUE){CONFIG, WRITABLE, SAUINT32T}<br>saAmfNodeRebootOnTerminationFailure : SaBoolT [0..1] = 0 (SA_FALSE){CONFIG, WRITABLE, SAUINT32T}<br>saAmfNodeRebootOnInstantiationFailure : SaBoolT [0..1] = 0 (SA_FALSE){CONFIG, WRITABLE, SAUINT32<br>saAmfNodeAdminState : SaAmfAdminStateT [1]{RUNTIME, CACHED, PERSISTENT, SAUINT32T}<br>saAmfNodeOperState : SaAmfOperationalStateT [1]{RUNTIME, CACHED, SAUINT32T} |
| SA_AMF_ADMIN_LOCK()<br>SA_AMF_ADMIN_SHUTDOWN()<br>SA_AMF_ADMIN_UNLOCK()<br>SA_AMF_ADMIN_LOCK_INSTANTIATION()<br>SA_AMF_ADMIN_UNLOCK_INSTANTIATION()<br>SA_AMF_ADMIN_RESTART()<br>SA_AMF_ADMIN_REPAIRED() |

## 4.7 AMF Application/SG Classes

| <<CONFIG>><br>SaAmfApplication |
|---|
| safApp : SaStringT [1]{ RDN, CONFIG}<br>saAmfApplicationAdminState : SaAmfAdminStateT [1]{RUNTIME, CACHED, PERSISTENT, SAUINT32T}<br>saAmfApplicationCurrNumSG : SaUint32T [1]{RUNTIME} |
| SA_AMF_ADMIN_LOCK()<br>SA_AMF_ADMIN_SHUTDOWN()<br>SA_AMF_ADMIN_UNLOCK()<br>SA_AMF_ADMIN_LOCK_INSTANTIATION()<br>SA_AMF_ADMIN_UNLOCK_INSTANTIATION()<br>SA_AMF_ADMIN_RESTART() |

| <<CONFIG>><br>SaAmfSG |
|---|
| safSg : SaStringT [1]{ RDN, CONFIG}<br>saAmfSGRedundancyModel : SaAmfRedundancyModelT [1]{CONFIG, SAUINT32T}<br>saAmfSGAutoAdjust : SaBoolT [0..1] = 0 (SA_FALSE){CONFIG, WRITABLE, SAUINT32T}<br>saAmfSGNumPrefActiveSUs : SaUint32T [0..1] = 1{CONFIG, WRITABLE}<br>saAmfSGNumPrefStandbySUs : SaUint32T [0..1] = 1{CONFIG, WRITABLE}<br>saAmfSGNumPrefInserviceSUs : SaUint32T [0..1] = Number of SUs{CONFIG, WRITABLE}<br>saAmfSGNumPrefAssignedSUs : SaUint32T [0..1] = saAmfSGNumPrefInserviceSUs{CONFIG, WRITABLE}<br>saAmfSGMaxActiveSIsperSUs : SaUint32T [0..1] = No limit{CONFIG, WRITABLE}<br>saAmfSGMaxStandbySIsperSUs : SaUint32T [0..1] = No limit{CONFIG, WRITABLE}<br>SaAmfSGCompRestartProb : SaTimeT [1]{CONFIG, WRITABLE}<br>SaAmfSGCompRestartMax : SaUint32T [1]{CONFIG, WRITABLE}<br>saAmfSGSuRestartProb : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfSGSuRestartMax : SaUint32T [1]{CONFIG, WRITABLE}<br>saAmfSGAutoAdjustProb : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfSGAutoRepair : SaBoolT [0..1] = 1 (SA_TRUE){CONFIG, WRITABLE, SAUINT32T}<br>saAmfSGAdminState : SaAmfAdminStateT [1]{RUNTIME, CACHED, PERSISTENT, SAUINT32T}<br>saAmfSGNumCurrAssignedSUs : SaUint32T [1]{RUNTIME}<br>saAmfSGNumCurrNonInstantiatedSpareSUs : SaUint32T [1]{RUNTIME}<br>saAmfSGNumCurrInstantiatedSpareSUs : SaUint32T [1]{RUNTIME} |
| SA_AMF_ADMIN_LOCK()<br>SA_AMF_ADMIN_SHUTDOWN()<br>SA_AMF_ADMIN_UNLOCK()<br>SA_AMF_ADMIN_LOCK_INSTANTIATION()<br>SA_AMF_ADMIN_UNLOCK_INSTANTIATION()<br>SA_AMF_ADMIN_SG_ADJUST() |

## 4.8 AMF SU Class

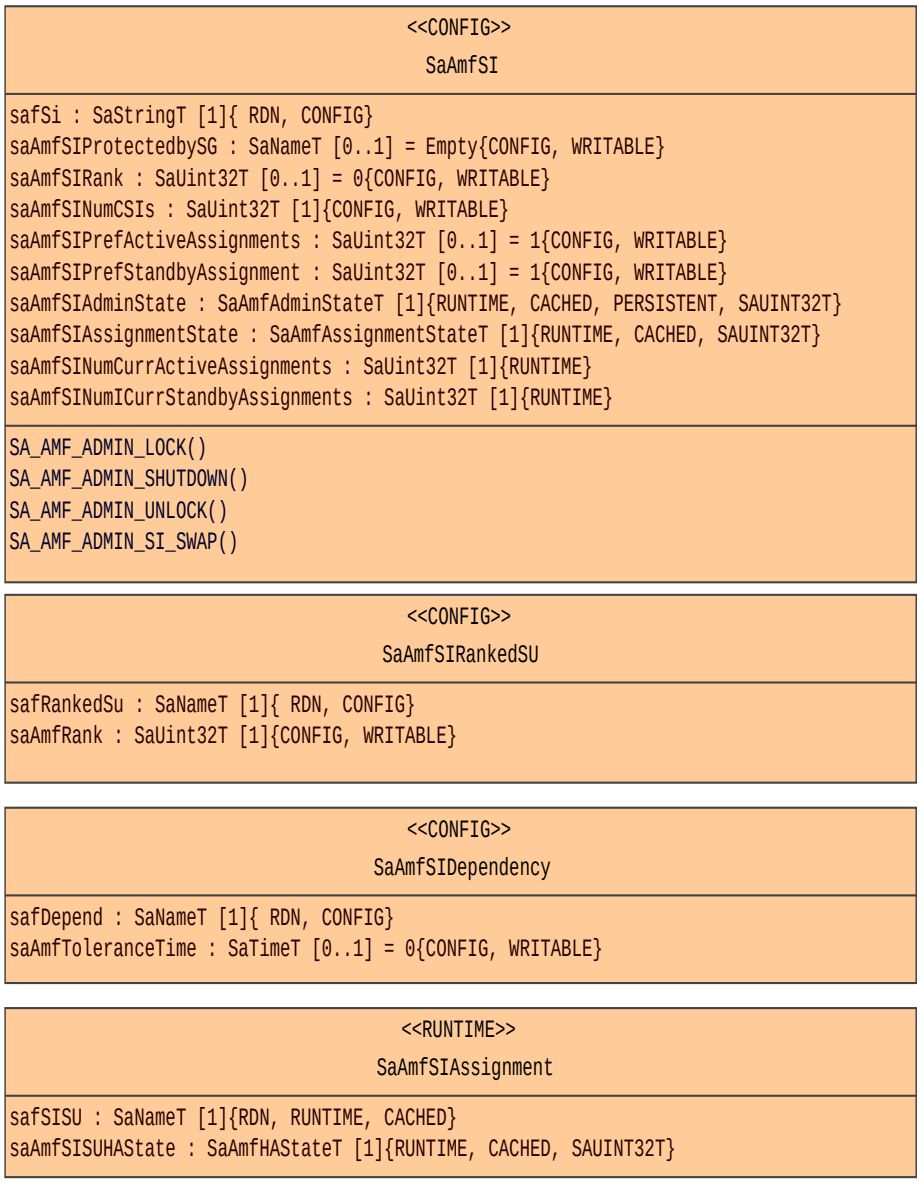


## 4.9 AMF Component

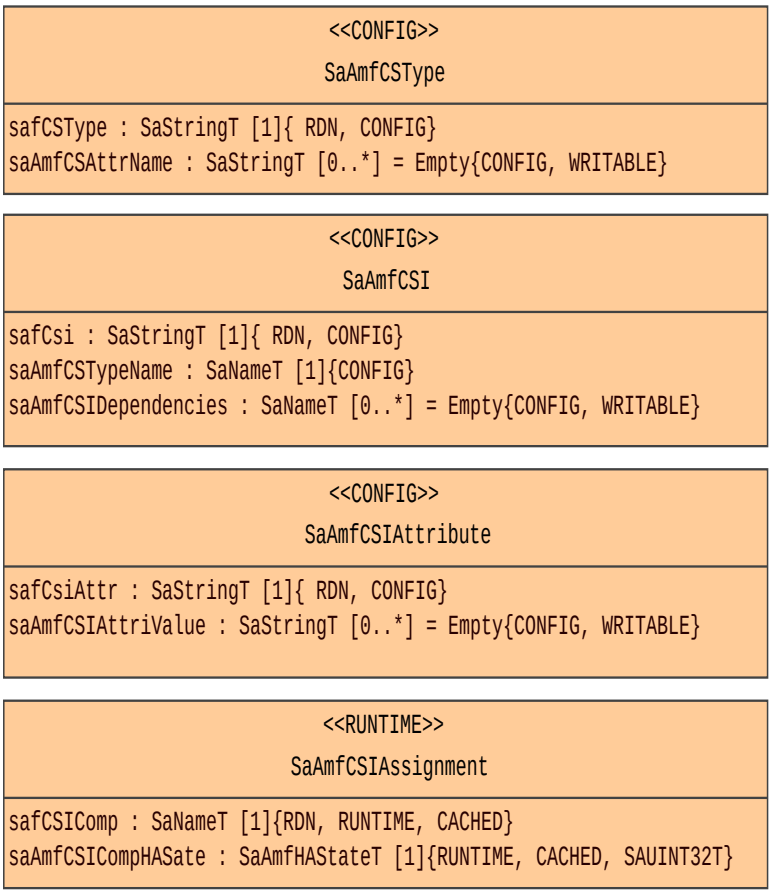
| <<CONFIG>><br>SaAmfComp |
|---|
| safComp : SaStringT [1]{ RDN, CONFIG}<br>saAmfCompCsTypes : SaNameT [1..*]{CONFIG, WRITABLE}<br>saAmfCompCategory : SaAmfCompCategoryT [1]{CONFIG, SAUINT32T}<br>saAmfCompCapability : SaAmfCompCapabilityModelT [1]{CONFIG, SAUINT32T}<br>saAmfCompNumMaxActiveCsi : SaUint32T [1]{CONFIG, WRITABLE}<br>saAmfCompNumMaxStandbyCsi : SaUint32T [1]{CONFIG, WRITABLE}<br>saAmfCompCmdEnv : SaStringT [0..*] = Empty{CONFIG, WRITABLE}<br>saAmfCompDefaultClcCliTimeout : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfCompDefaultCallbackTimeOut : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfCompInstantiateCmd : SaStringT [0..1] = Empty{CONFIG, WRITABLE}<br>saAmfCompInstantiateCmdArgv : SaStringT [0..*] = Empty{CONFIG, WRITABLE}<br>saAmfCompInstantiateTimeout : SaTimeT [0..1] = saAmfCompDefaultClcCliTimeout{CONFIG, WRITABLE}<br>saAmfCompInstantiationLevel : SaUint32T [0..1] = 0{CONFIG, WRITABLE}<br>saAmfCompNumMaxInstantiateWithoutDelay : SaUint32T [0..1] = 2{CONFIG, WRITABLE}<br>saAmfCompNumMaxInstantiateWithDelay : SaUint32T [0..1] = 0{CONFIG, WRITABLE}<br>saAmfCompDelayBetweenInstantiateAttempts : SaTimeT [0..1] = 0{CONFIG, WRITABLE}<br>saAmfCompTerminateCmd : SaStringT [0..1] = Empty{CONFIG, WRITABLE}<br>saAmfCompTerminateTimeout : SaTimeT [0..1] = saAmfCompDefaultClcCliTimeout{CONFIG, WRITABLE}<br>saAmfCompTerminateCmdArgv : SaStringT [0..*] = Empty{CONFIG, WRITABLE}<br>saAmfCompCleanupCmd : SaStringT [1]{CONFIG, WRITABLE}<br>saAmfCompCleanupCmdArgv : SaStringT [0..*] = Empty{CONFIG, WRITABLE}<br>saAmfCompCleanupTimeout : SaTimeT [0..1] = saAmfCompDefaultClcCliTimeout{CONFIG, WRITABLE}<br>saAmfCompAmStartCmd : SaStringT [0..1] = Empty{CONFIG, WRITABLE}<br>saAmfCompAmStartCmdArgv : SaStringT [0..*] = Empty{CONFIG, WRITABLE}<br>saAmfCompAmStartTimeout : SaTimeT [1] = saAmfCompDefaultClcCliTimeout{CONFIG, WRITABLE}<br>saAmfCompNumMaxAmStartAttempt : SaUint32T [0..1] = 2{CONFIG, WRITABLE}<br>saAmfCompAmStopCmd : SaStringT [0..1] = Empty{CONFIG, WRITABLE}<br>saAmfCompAmStopCmdArgv : SaStringT [0..*] = Empty{CONFIG, WRITABLE}<br>saAmfCompAmStopTimeout : SaTimeT [0..1] = saAmfCompDefaultClcCliTimeout{CONFIG, WRITABLE}<br>saAmfCompNumMaxAmStopAttempt : SaUint32T [0..1] = 2{CONFIG, WRITABLE}<br>saAmfCompTerminateCallbackTimeout : SaTimeT [0..1] = saAmfCompDefaultCallbackTimeout{CONFIG, WRITABLE}<br>saAmfCompCSISetCallbackTimeout : SaTimeT [0..1] = saAmfCompDefaultCallbackTimeout{CONFIG, WRITABLE}<br>saAmfCompQuiescingCompleteTimeout : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfCompCSIRmvCallbackTimeout : SaTimeT [0..1] = saAmfCompDefaultCallbackTimeout{CONFIG, WRITABLE}<br>saAmfCompRecoveryOnError : SaAmfRecommendedRecoveryT [1]{CONFIG, WRITABLE, SAUINT32T}<br>saAmfCompDisableRestart : SaBoolT [0..1] = 0 (SA_FALSE){CONFIG, WRITABLE, SAUINT32T}<br>saAmfCompProxyCsi : SaNameT [0..1] = Empty{CONFIG, WRITABLE}<br>saAmfCompOperState : SaAmfOperationalStateT [1]{RUNTIME, CACHED, SAUINT32T}<br>saAmfCompReadinessState : SaAmfReadinessStateT [1]{RUNTIME, CACHED, SAUINT32T}<br>saAmfCompPresenceState : SaAmfPresenceStateT [1]{RUNTIME, CACHED, SAUINT32T}<br>saAmfCompRestartCount : SaUint32T [1]{RUNTIME}<br>saAmfCompNumCurrActiveCsi : SaUint32T [1]{RUNTIME}<br>saAmfCompNumCurrStandbyCsi : SaUint32T [1]{RUNTIME}<br>saAmfCompAssignedCsi : SaNameT [0..*] = Empty{RUNTIME}<br>saAmfCompCurrProxyName : SaNameT [1]{RUNTIME}<br>saAmfCompCurrProxiedNames : SaNameT [0..*] = Empty{RUNTIME} |
| SA_AMF_ADMIN_RESTART()<br>SA_AMF_ADMIN_EAM_START()<br>SA_AMF_ADMIN_EAM_STOP() |

| <<CONFIG>><br>SaAmfHealthcheck |
|---|
| safHealthcheckKey : SaAmfHealthcheckKeyT [1]{RDN,CONFIG,SASTRINGT}<br>saAmfHealthcheckPeriod : SaTimeT [1]{CONFIG, WRITABLE}<br>saAmfHealthcheckMaxDuration : SaTimeT [1]{CONFIG, WRITABLE} |

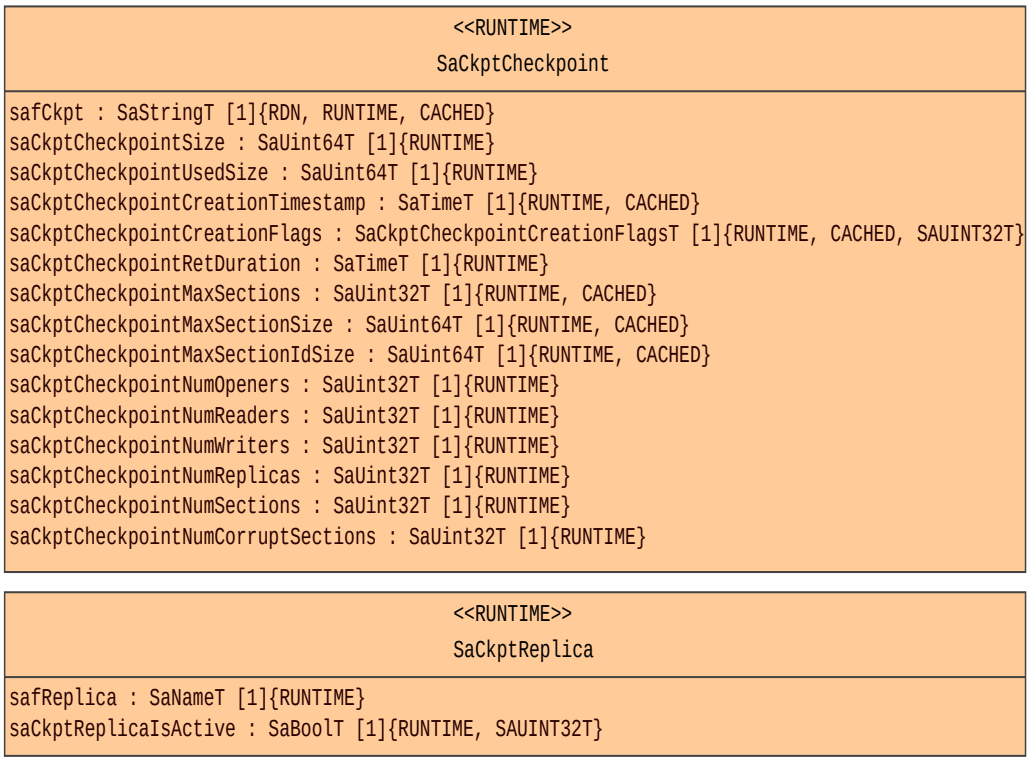
## 4.10 AMF SI Classes



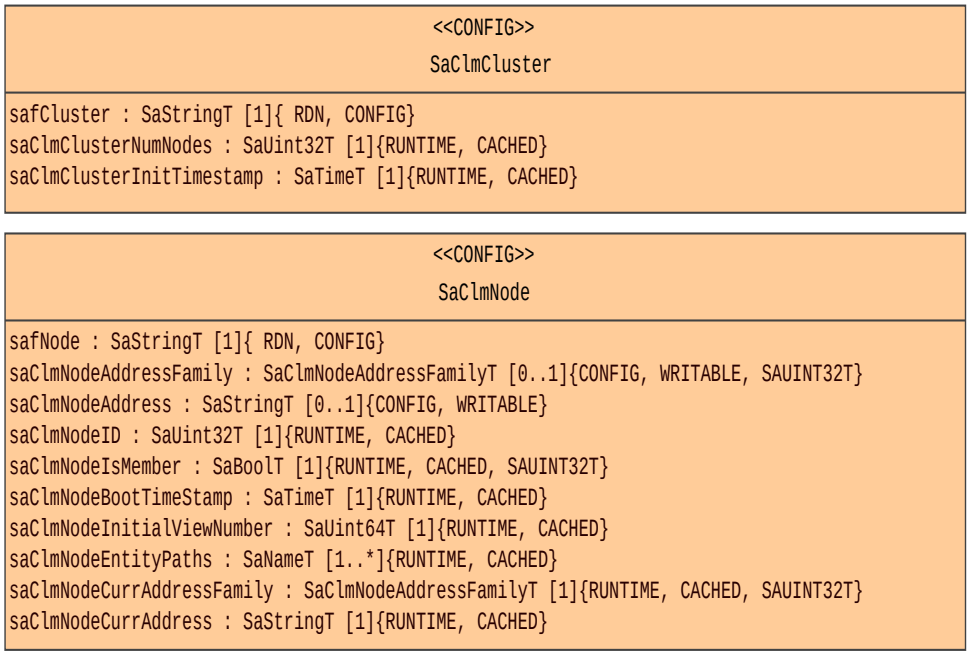
## 4.11 AMF CSI Classes



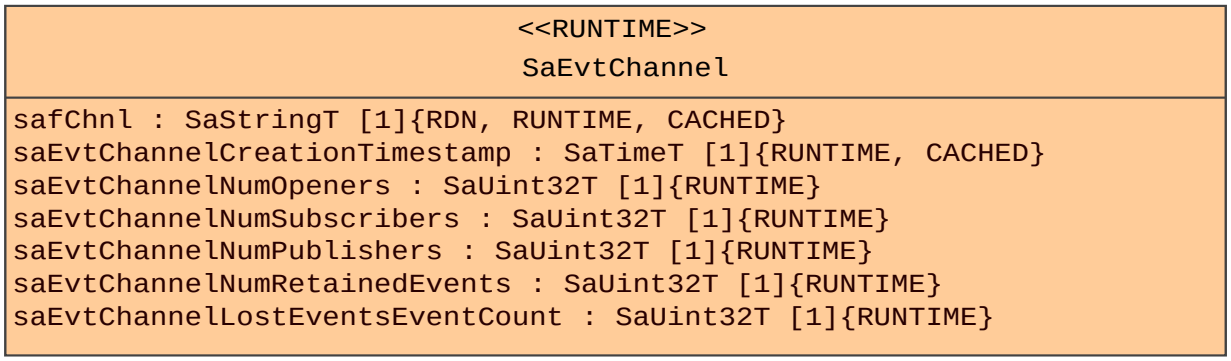
## 4.12 CKPT Classes



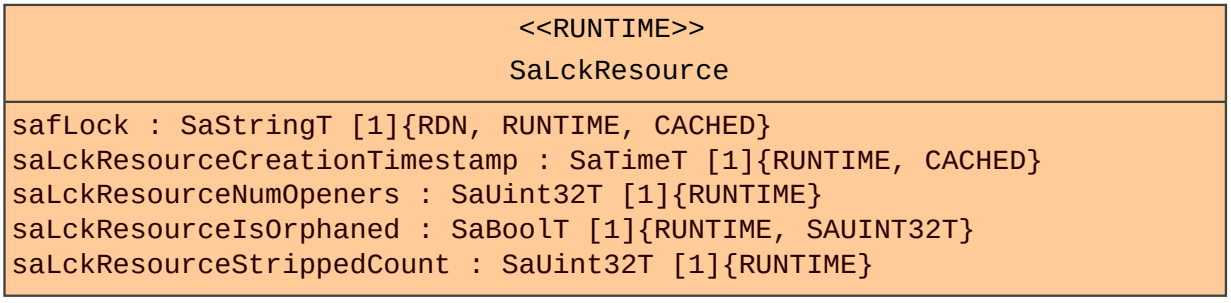
## 4.13 CLM Classes



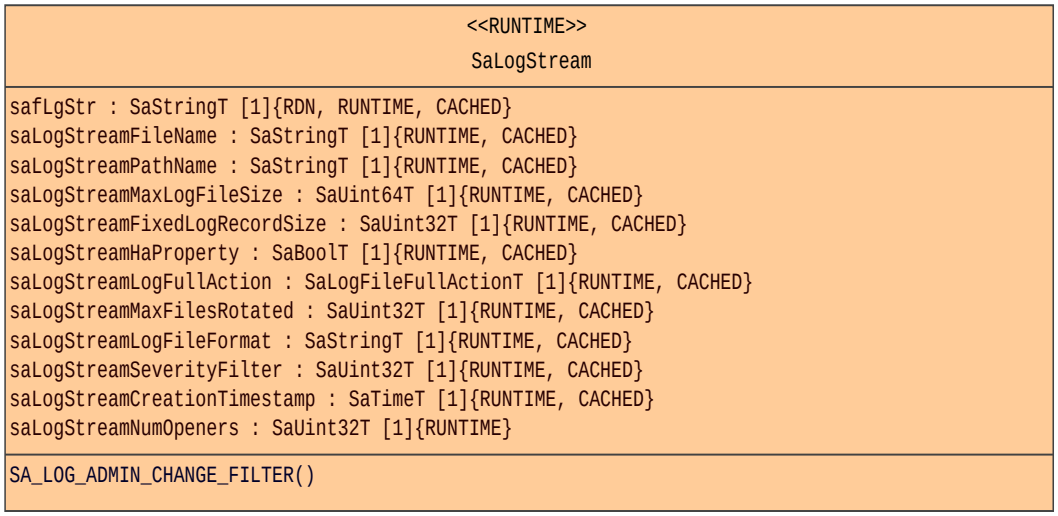
## 4.14 EVT Classes

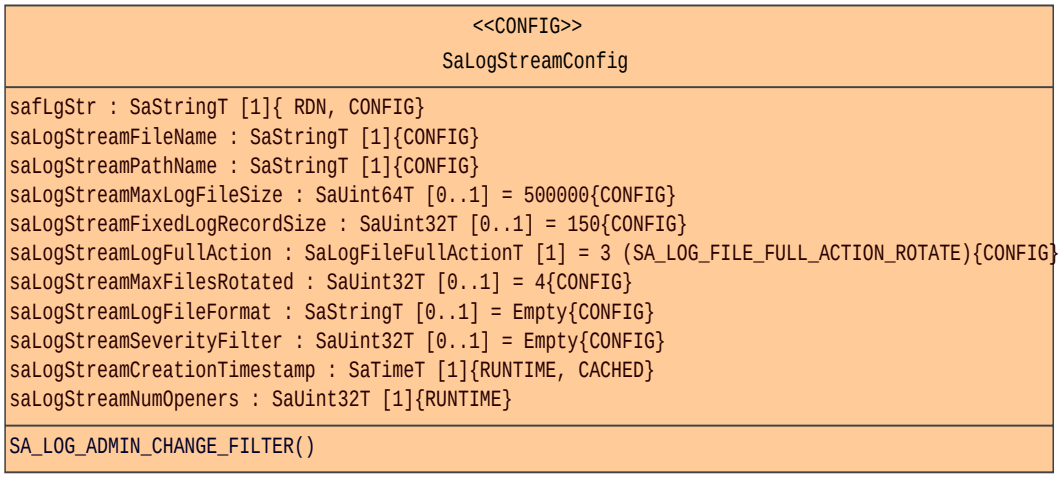


## 4.15 LCK Classes

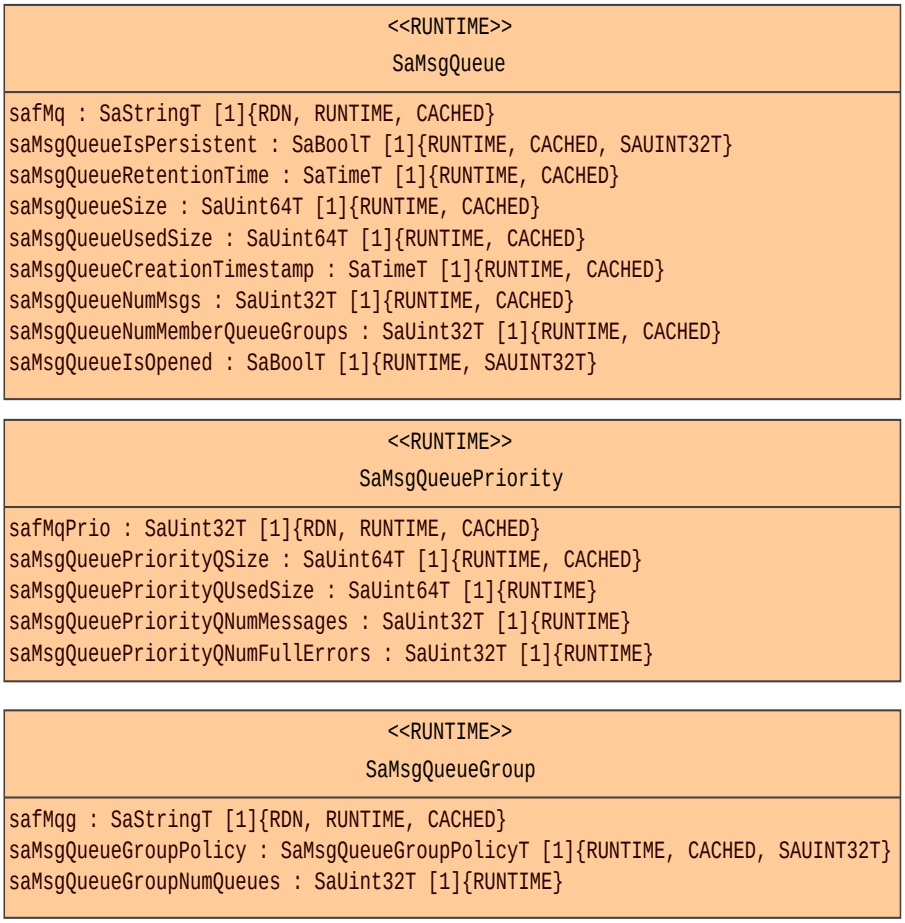


## 4.16 LOG Classes



## 4.17 MSG Classes



# 5 AIS Abbreviations, Concepts, and Terminology

This chapter presents the main abbreviations, concepts and terms used in the SA AIS documents. A reasonable understanding of basic computing and high availability terminology is assumed of the reader.

### Active

Providing a service.

### Active HA State

This is one of the HA states that can be assumed by a component or a service unit. Refer to “HA State”.

### Active Replica

An active replica is only defined for checkpoints created with the asynchronous update option. For these checkpoints, checkpoint data is read from an active replica, and checkpoint data is first written synchronously to an active replica and asynchronously replicated to other replicas.

### Active Service Unit

This is a service unit having the active HA state for all service instances assigned to it.

### Administrative State

The administrative state is manipulated by the system administrator and refers to the ITU X.731 state management model (see [2]). It is used by the Availability Management Framework to determine whether a service unit, a service group, a service instance, a node, or the cluster are administratively allowed to provide the service. Valid values of the administrative state are unlocked, locked, locked-instantiation, and shutting-down.

### Application

AIS defines an application as a logical entity that contains one or more service groups. An application combines the individual functionalities of the constituent service groups to provide a higher level service. This aggregation provides the Availability Management Framework with a further scope for fault isolation and fault recovery.

From a software administration point of view, this grouping into application reflects the set of service units and contained components that are delivered as a consistent

set of software packages, which results in tighter dependency with respect to their upgrade.

**Area**

A functionality specified by the SA Forum (Availability Management Framework, Cluster Membership Service, etc.).

**Area Server**

The area server is an abstraction that represents the server that provides services for a specification area (Availability Management Framework, Cluster Membership Service, etc.).

**Assigned Service Unit**

These are service units that have at least one service instance assigned to them.

**Assignment**

Besides its normal use in English, the term assignment is used in the following constructions in the description of the Availability Management Framework:

- Assignment of a service instance to a service unit
- Assignment of a component service instance to a component
- Assignment of an HA state to a service unit for a service instance
- Assignment of an HA state to a component for a component service instance
- HA state assignment of/for a service instance, or simply HA state assignment, if the context makes it clear which service instance is meant.
- HA state assignment of/for a component service instance, or simply HA state assignment, if the context makes it clear which component service instance is meant.

Refer to the description of the Availability Management Framework for details.

**Assignment State**

This state is defined for service instances. It 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", "fully-assigned" and “partially-assigned”.

### Asynchronous Checkpoint Update

When a checkpoint has been created with the asynchronous update option, write calls return immediately when the active checkpoint replica has been updated. Other replicas are updated asynchronously. To guarantee that a process does not read stale data, the SA Checkpoint Service always reads from an active checkpoint replica. There are two variants of the asynchronous update: For the first one, either a replica is updated with all data of a write call or not at all; the second variant does not provide the atomicity guarantee of the first one, but the SA Checkpoint Service marks sections that are modified by the write call as corrupt when a fault occurs while a checkpoint replica is being updated.

### Auto-Adjust

This notion indicates that it is required that the service instance assignments to the service units in a service group are transferred back to the most preferred service assignments in which the highest ranked available service units are assigned the active or standby HA states for those service instances. 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.

### Auto Repair

A configuration attribute at node and service group level that determines if the Availability Management Framework engages in automatic repair or not.

### Availability Management Framework

The Availability Management Framework is the software entity that manages the resources in the system to deliver availability. The Availability Management Framework provides a view of one logical cluster consisting of a number of cluster nodes. The Availability Management Framework provides availability by coordinating redundant resources within the cluster to deliver a system with no single point of failure.

### Checkpoint

The Checkpoint Service provides a facility for processes to record checkpoint data incrementally, which can be used to protect an application against failures. When processes restart, or a set of other processes take over the work of the failed processes (through a fail-over procedure), the Checkpoint Service can be used to restore the saved checkpoint data and resume execution from the state recorded before the failure, minimizing the impact of the failure.

### Checkpoint Replica

A checkpoint replica is a copy of the data that is stored in a checkpoint. At most one checkpoint replica for a particular checkpoint may reside on a given node. A given checkpoint may have several checkpoint replicas (copies) on different nodes.

### Checkpoint Section

Each checkpoint is structured as a set of sections that can be created and deleted dynamically by processes. Each section is identified by a unique identifier, the section identifier. Sections contain raw data provided by processes. Sections have an expiration time.

### Checkpoint Service

The Checkpoint Service is an SA Service that provides a facility for a process to retrieve and transfer its state on one or more nodes. Recording the state of a process in checkpoints protects the application against failure of the process. To recover from the failure of a process, the process taking over the service can retrieve the previous checkpoint from the SA Checkpoint Service and resume execution from that checkpoint recorded before the failure, minimizing the impact of the failure on the process's clients.

### CLC

CLC is an acronym for Component Life Cycle. It is used in the Availability Management Framework for instantiating, terminating, cleaning up and actively monitoring local components.

### CLC-CLI

In the context of the Availability Management Framework, this is a set of command line interfaces (CLI) provided by local components to enable the Availability Management Framework to control the life cycle of these components.

### CLI

CLI is an acronym for Command Line Interfaces.

### Cluster

A cluster is a collection of cluster nodes (see "Cluster Node") that may change dynamically as nodes join or leave the cluster.

**Cluster Membership**

The cluster membership is the set of cluster nodes that take part in the cluster. A membership transition is a change in the membership of the cluster. A “view number” is associated with each membership transition.

**Cluster Membership Service**

The Cluster Membership Service is an SA Service that provides processes a means of retrieving information about the cluster membership and the cluster member nodes. It also enables a process to register a callback function to receive membership change notifications as they occur.

**Cluster Node**

A cluster node is the logical representation of a physical node (see “Physical Node”).

**Cluster-Wide**

The attribute 'cluster-wide' is used to indicate logical entities that span one or more nodes and that are designated by names unique in the entire cluster. Depending on the particular entity, it can be accessible from all nodes of the cluster or only from a subset of the nodes of the cluster.

**Collocated Checkpoint**

A collocated checkpoint is a type of checkpoint for which the application has full responsibilities for replica creation and for deciding which replica is the active one. The replicas of a collocated checkpoint are only created by the applications via open calls, provided that no local replica already exists.

**Component**

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.

**Component Healthcheck Monitoring**

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. Healthchecks can be invoked

by the Availability Management Framework (Framework-invoked healthchecks) or by the component (component-invoked heathchecks). It is up to the component to start and terminate the healthcheck monitoring. See also “Component Monitoring”.

### Component Capability Model

To accommodate possible simplifications in component development, whereby components may implement only restricted capabilities, the Availability Management Framework defines the Component Capability Model. For details, refer to the Availability Management Framework specification.

### Component Monitoring

The Availability Management Framework supports three types of component monitoring:

- 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 errors. This internal monitoring code is invoked periodically to assess the component health from inside the component. This type of monitoring is also known as “component healthcheck monitoring”.

### Component Service Instance

A component service instance is a particular service (workload) that the Availability Management Framework has assigned to a component in a service unit. See “Service Instance”.

### 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.

### Component State

The overall state of a component is a combination of a number of underlying states. Each component in a service unit has presence, operational and readiness states, as well as an HA state on behalf of each component service instance assigned to the component.

### Component Registration

A component registers with the Availability Management Framework to inform the Availability Management Framework that it is ready to provide service for component service instances by receiving assignment of HA states for these component service instances. After registration, the Availability Management Framework can invoke call-back functions provided by the component.

### Configuration Objects and Attributes

Configuration objects and attributes are the means by which system management applications provide input on the desired sets of objects and their handling that an Object Implementer should implement. The set of configuration objects and attributes constitute the prescriptive part of the information model.

### CSI

CSI is an acronym for component service instance.

### Disabled State

This is one of the operational states that can be assumed by a component, a service unit, or a cluster node. Refer to the description of the Availability Management Framework for more details.

### Enabled State

This is one of the operational states that can be assumed by a component, a service unit, or a cluster node. Refer to the description of the Availability Management Framework for more details.

### Error

Incorrect information provided by a component or process, or the lack of correct information at the correct time, where such incorrect information or lack of information will lead to a failure in the absence of error detection and recovery mechanisms.

### 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. 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.

### Event

An event consists of a set of event attributes (sometimes called the event header) and zero or more bytes of event data. Refer to “Event Attributes” and “Event Service” for more details.

### Event Attributes

These are event pattern array, event priority, event publish time, event retention time, event publisher name, and event id.

### Event Filters

These are filters, which are used by an application process to specify the events the application is interested in. If the filters match the event patterns, the application process is notified of these events. See also “Event Pattern”.

### Event Channel

An event channel is a mechanism provided by the SA Event Service for publishers and subscribers to communicate via events. A process can open the event channel to publish events and to subscribe to receive events. Publishers can also be subscribers on the same event channel.

### Event Pattern

An event pattern is an event attribute used to organize events into various categories. An event pattern can be used for filtering out the events that a subscriber is not interested it. All publishers/subscribers of an event channel must share the same conventions regarding the number of patterns being used, their ordering and contents, as well as meaning.

### Event Publisher

A process that publishes events on an event channel.

**Event Service**

The Event Service is an SA Service that provides a publish/subscribe communication mechanism based on event channels and asynchronous communication between publishers and subscribers. Subscribers are anonymous, which means that they may join and leave an event channel at any time without involving the publisher(s). The primary usage of the SA Event Service is the reporting of generic events in the system.

**Event Subscriber**

A process that is interested in receiving published events on a specific event channel.

**External Active Monitoring**

This is a type of component monitoring supported by the Availability Management Framework. 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.

**External Component**

An external component represents a set of resources that are external to the cluster nodes.

**External Resource**

Any resource that is not a local resource, i.e., that is not contained within a physical node.

**External Service Unit**

A service unit containing only external components.

**Fail-over**

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.

### Failure

The event of a component or a service unit not providing service at the level that the component or service unit is required to provide.

### Fault

The cause of an error, typically a failure of a component or process.

### 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, 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.

### HA State

This state is maintained at the component level on behalf of a component service instance, and at service unit level on behalf of a service instance. It can take the values: active, standby, quiescing, or quiesced.

Refer to the description of the Availability Management Framework for more details.

### Healthcheck Key

The key of the healthcheck to be executed. Using this key, the Availability Management Framework can retrieve the corresponding healthcheck parameters.

### Information Model

The SA Forum information model (IMM) is specified in UML and represents through its objects the various entities, which constitute an SA Forum system. The SA Forum IM also specifies the attributes of these objects and administrative operations that can be performed on the entities through system management interfaces.

### IMM

IMM is an acronym for Information Model Management.

**Information Model Management Service**

The Information Model Management (IMM) Service is the SA service managing all objects of the SA Information Model and provides the APIs to access and manage these objects.

**In-Service Service Unit**

This is a service unit that has a readiness state of either in-service or stopping.

**In-Service State**

This is one of the readiness states. Refer to “Readiness State”.

**Instantiable Service Unit**

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.
- The service unit’s presence state is uninstantiated.
- The service unit’s instantiation has not been administratively prevented. Note that this is different from the lock/unlock administrative operations.

**Instantiated Service Unit**

These are service units with the presence state of either instantiated or restarting.

**Internal Active Monitoring**

Refer to “Component Healthcheck Monitoring”.

**Local Component**

A local component represents a subset of the local resources contained within a single physical cluster node.

**Local Replica**

This is a checkpoint replica located on the node where the checkpoint is opened.

**Local Resource**

A resource that is contained, from a fault containment point of view, within a physical node. 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.

### Local Service Unit

This is a service unit containing only local components.

### Lock

A lock is a protected access to a lock resource requested through the SA Lock Service. See also “Lock Resource”. Locks can be requested in either exclusive or shared read mode.

### Lock Administrative Operation

This operation applies to a service unit, a service group, a cluster node, a service instance, an application, and the cluster. It sets the administrative state of a logical entity to locked, which results in the removal of workload from the logical entity.

### Lock-Instantiation Administrative Operation

This operation applies to a service unit, a service group, a cluster node, a service instance, an application, and the cluster. It sets the administrative state of a logical entity to locked-instantiation, which results in the termination of this logical entity.

### Locked State

One of the administrative states that are maintained for service units, service groups, cluster nodes, service instances, applications, and the cluster. An entity in this state has been administratively prevented from participating in providing service. See also “Lock Administrative Operation”.

### Locked-instantiation State

One of the administrative states that are maintained for service units, service groups, cluster nodes, service instances, applications, and the cluster. An entity in this state has been administratively prevented from being instantiated. See also “Lock-Instantiation Administrative Operation”.

### Locked Service Unit

In the description of the Availability Management Framework, saying that a service unit is locked without further specification implies that its administrative state or one of the administrative states of the following entities is locked: service group containing it, cluster node containing it (for local service units), enclosing application (if any), and cluster (for local service units).

**Lock Mode**

The SA Lock Service supports two lock modes:

- Protected Read (PR) - A shared read, i.e., any number of lock owners may hold a read lock and no one may hold an exclusive lock.
- Exclusive (EX) - Only a single lock owner may hold the lock.

**Lock Owner**

The process holding a lock in the exclusive mode or one or more processes holding a lock in the shared read mode. Refer to “Lock” and “Lock Service” for details.

**Lock Resource**

A lock resource is an entity, supported by the SA Lock Service, that is used to synchronize access to shared resources between application processes.

**Lock Service**

The Lock Service is an SA Service that provides entities, called lock resources, that synchronize access to shared resources between application processes.

**Log Stream**

A conceptual flow of log records that travel from any number of sources to a destination output such as a log file.

**Log Record**

A log record is the unit of thing logged by some process called a Logger.

**Log Filter**

A log filter specifies criteria for determining if a log record is allowed on a log stream. These criteria are based on severity level values.

**Log Record Output Formatting Rules**

Log record output formatting rules consist of a well-known set of log record format tokens that can be ordered into well-formed log record format expressions. The log record output format rules govern the output properties of each log record at an output destination.

**Logical Node**

See “Cluster Node”.

**Message**

A message is a byte array of a certain size. In addition to the data contained in the byte array, a message also has a type, version and priority.

**Message Queue**

A message queue is a software abstraction for buffering messages. A message queue consists of a collection of separate data areas that are used to store messages of different priorities. Messages are read from, and written to, a message queue.

**Message Queue Group**

A message queue group is a collection of message queues that are addressed as a single entity. A message queue group is global to a cluster, and it is identified by a unique name in the same name space used for the names of message queues. A message queue can be a member of different message queue groups. Messages sent to a message queue group are directed to one or more of the message queues in the group. See also “Unicast Message Queue Group” and “Multicast Message Queue Group”.

**Message Service**

The Message Service is an SA Service that provides a buffered message passing system based on the concept of a message queue and that also supports load balancing via message queue groups.

**Multicast Message Queue Group**

A multicast message queue group is one such that a message can be sent to multiple message queues in the group simultaneously.

**Node**

See “Cluster Node”.

**Node Id**

Node identifier that is unique in the cluster. A given node id identifies a node only for as long as the node is a member of the cluster membership.

**Node Name**

The name of a node that is unique in the cluster.

### Non-Collocated Checkpoints

Checkpoints created without the collocated attribute are called non-collocated checkpoints. The management of replicas of non-collocated checkpoints and whether they are active or not is mainly the duty of the SA Checkpoint Service.

### 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-instantiatable components.

### Non-Pre-Instantiable Service Unit

This is a service unit containing no pre-instantiable component.

### Non-SA-Aware Component

A non-SA-aware component is a component that provides no specific support for SA functionality provided by the Availability Management Framework. Such components do not use any Availability Management Framework API.
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 non-SA-aware components for which a proxy component mediates are called proxied 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 via CLC-CLI commands.

### Notification

In the context of the Notification Service, notifications are data containers informing consumers about an event that has occurred. There are various notification types defined. A set of notification attributes is common to all notification types. Additionally, there are also attributes specific to the notification types.

### Notification Consumer

In the context of the Notification Service, notification consumers are notification readers or subscribers.

### Notification Filter

Notification filters can be specified for the operations of the reader API and subscriber API of the Notification Service.

### Notification Producer

A notification producer generates notifications using the producer API of the Notification Service.

### Notification Reader

A notification reader retrieves persistently logged notifications using the reader API of the Notification Service.

### Notification Service

The Notification Service is an SA Service that allows notifications to be produced and consumed. In contrast to the Event Service, the Notification Service has the following characteristics:

Notification types and attributes of the notifications are closely related to objects defined in the ITU-T X.73x recommendations.

The Notification Service provides guaranteed delivery for notifications forwarded to subscribers.

### Notification Subscriber

A notification subscriber gets notifications forwarded as they occur using the subscriber API of the Notification Service.

### Notification Type

The Notification Service defines the following notification types:

- object creation and deletion,
- attribute change,
- state change,
- alarm, and
- security alarm.

### Object Implementer

Services and applications that implement the IMM objects are called Object Implementers. This is a term used in the SA IMM Service.

**Object Implementer API**

The Object Implementer API (OI-API) is defined in the SA IMM Service, and its use is restricted to Object Implementers.

**Object Management API**

The Object Management API (OM-API) is defined in the SA IMM Service and exposed typically to system management applications.

**Operational State**

The operational state is supported for components, service units, and cluster nodes. It is used by the Availability Management Framework to determine whether one of these entities is capable of taking a role in providing the service. The operational states are enabled and disabled.

**Orphan Locks**

These are locks that are held by a process when the process terminates. Usually, the SA Lock Service strips the orphan locks from the holding process. However, for SA Lock Service implementations supporting the orphan lock feature, the user can disable the stripping of locks. Orphan locks need to be freed with a purge call.

**Out-of-Service State**

This is one of the readiness states. Refer to “Readiness State”.

**Partial Checkpoint Update**

A checkpoint with the partial update option is a checkpoint with the asynchronous update option and with no guarantee for the atomicity of an update. The SA Checkpoint Service marks sections that are modified by the write call as corrupt when a fault occurs while a checkpoint replica is being updated. For an asynchronous checkpoint without the partial update option, it is guaranteed that a replica is either updated with all data specified in the write call or is not updated at all.

**Partially-Assigned**

A configured service instance that is neither unassigned nor fully-assigned is said to be partially-assigned.

**Passive Monitoring**

This is a type of component monitoring supported by the Availability Management Framework. Refer to “Component Monitoring” for details.

### Persistent and Non-Persistent Message Queues

A message queue can be defined as non-persistent, meaning that the SA Message Service removes the message queue automatically from the cluster-wide name space if it is not open by a process for a configurable amount of time, called the retention time. The retention time starts after a static creation or after a close.

A persistent message queue is like a non-persistent message queue with an infinite retention time. Thus, its life cycle is not tied to the life cycle of the process that created it. A persistent message queue can only be removed by using the *saMsgQueueUnlink()* call.

### Pre-Instantiable Component

Such a component has the ability to stay idle when it gets instantiated by the Availability Management Framework. It only starts to provide a particular service when instructed to do so (directly or indirectly) by the Availability Management Framework.

### Pre-Instantiable Service Unit

This is a service unit that contains at least one pre-instantiatable component.

### Presence State

The presence state is supported at the component and service unit levels and reflects the component life cycle. Its values are: Uninstantiated, Instantiating, Instantiated, Terminating, Restarting, Instantiation-failed, Termination-failed.

### Process

The term process, in this document, can be regarded as equivalent to a process defined by the POSIX standard. However, use of the term process does not mandate a POSIX process, but rather any equivalent entity that a system provides to manage executing software.

### Protection Group

A protection group for a component service instance is the group of components 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 as component service instances are assigned to, and removed from, components.

**Proxy Component**

This is an SA-aware component that mediates for non-SA-aware components. A proxy component registers the non-SA-aware component, called proxied component, with the Availability Management Framework and receives instructions from the Availability Management Framework on behalf of the proxied component.

**Proxied Component**

This is a non-SA-aware component that is mediated by a proxy component. It does not interact directly with the Availability Management Framework via API interfaces; requests from the Availability Management Framework for proxied components are directed to the proxy component acting on behalf of the proxied component.

**Quiesced HA State**

This is one of the HA states that can be assumed by a component or a service unit. Refer to “HA State”.

**Quiescing HA State**

This is one of the HA states that can be assumed by a component or a service unit. Refer to “HA State”.

**Readiness State**

This state is applicable to service units and components.

At the service unit level, this is a compound state determined by the administrative and operational state of the service unit as well as the administrative state of the service group containing it, cluster node containing it (for local service units), enclosing application (if any), and cluster (for local service units).

At the component level, this is a compound state that is a combination of the enclosing service unit's readiness state and the component's own operational state.

The valid values for the readiness state are in-service, out-of-service and stopping.

Readiness is the one and only state that determines the ability of service units to receive service instance assignments and the ability of components to receive component service instance assignments.

**Recommended Recovery**

The type of recovery recommended to the Availability Management Framework to recover from an error of a component. The recovery taken by the Availability Management Framework is not necessarily the recovery recommended by the error

report, because the Availability Management Framework may decide to escalate to a higher recovery level.

### 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. Note that the Availability Management Framework may have to (or chooses to) reassign these component service instances to non-erroneous components with a different HA state than the HA state of the erroneous component. Recovery actions include different levels of restart and different levels of fail-over.

### Recovery Escalation

When an error is reported on a component, the error report also contains a recommended recovery action. The Availability Management Framework, by default, implements the recommended recovery action; however, to cover cases in which the recovery action is not appropriate and does not prevent the error from occurring again, the Availability Management Framework also implements an escalation recovery policy. The underlying principle of the escalation is to progressively extend the scope of the error from component to service unit, from service unit to node and from node to application or cluster.

### Redundancy Level of a Service Instance

The number of service units being assigned an HA state for this service instance.

### Redundant Service Unit

A redundant service unit is a service unit that can be used if the original service unit fails.

### Registered Process

The process of an SA-aware component that is linked to the Availability Management Framework library, and that registered a component with the Availability Management Framework. The registered component can be the component the process belongs to or a proxied component. The registered process executes the requests of the Availability Management Framework on behalf of the component and conveys such requests to other processes and the hardware equipment of the local component, where necessary.
Note that the "registered" refers to the component and not to the process.

**Repair**

This is the action performed by the Availability Management Framework to bring some erroneous components back into a healthy state. Note that in case of HW, repair may require human intervention (to replace a board for example).
Repair actions include restarting a component, restarting all components in a service unit, and rebooting a node.
Note that a restart action is both a recovery and a repair action. Sometimes, restart is used only as a repair action. For example, if the recovery action is to fail over the erroneous component, the repair action can be to restart the component.

**Resource**

Any physical entity managed by the Availability Management Framework is a resource. 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.

**Restart**

Restarting a component means that the component is terminated and then reinstantiated.

**Runtime Objects and Runtime Attributes**

Runtime objects and attributes are the means by which Object Implementers reflect the current state of the objects they implement in the information model. The set of runtime objects and attributes constitute the descriptive part of the information model. Runtime objects and attributes are typically under the control of Object Implementers.

**SA-Aware Component**

An SA-aware component is a local component that supports the SA functionality provided by the Availability Management Framework and is under direct control of the Availability Management Framework. The Availability Management Framework directs operations to the component as to what it should do and when to do it. The component then reacts appropriately to these operations.

**Service Availability Services**

Service Availability (SA) Services are components of this specification that provide core underlying services for the Availability Management Framework and applications.

### Service Group

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, a service unit must support the **redundancy model** (see below) defined for the service group and be able to receive the assignment of any service instance protected by the service group.

### 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.

This specification defines the following service group redundancy models:

- 2N redundancy model: 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 are considered spare service units for the service group. 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.
- N+M redundancy model: 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 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.
- N-Way redundancy model: In a service group with the N-way redundancy model, a service unit can simultaneously
  (i) be assigned 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.
- N-Way Active redundancy model: 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.

- "No Redundancy" redundancy model: 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 containing the service unit.
  - No two service units exist having the same SI assigned to them.

**Service Instance**

A service instance is a particular service (workload) that the Availability Management Framework has assigned to a service unit. A service instance is made up of a set of component service instances that correspond to the service assigned to each component in the service unit. A single service unit may support multiple service instances at a given time.

**Service Unit**

A service unit 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. From the Availability Management Framework's perspective, a service unit is the unit of redundancy. A service unit can comprise any number of components, but a given component can exist in only one service unit at a time. External and local components cannot be mixed in the same service unit.

**Service Unit State**

The overall state of a service unit is a combination of a number of underlying states. Each service unit has presence, administrative, operational and readiness states, as well as an HA state on behalf of the service instances assigned to the service unit.

**SG**

SG is an acronym for service group.

**Shutting Down Administrative Operation**

This operation is derived from the ITU X.731 state management model (see [2]). It applies to a service unit, a service group, a cluster node, a service instance, an application, and the cluster.

**SI**

SI is an acronym for service instance.

**Spare Service Unit**

This is an instantiated service unit having no service instance assigned to it.

**Standby**

Not currently providing service but prepared to take over the active state.

**Standby HA State**

This is one of the HA states that can be assumed by a component or a service unit. Refer to “HA State”.

**Standby Service Unit**

This is a service unit having the standby HA state for all service instances assigned to it.

**SU**

SU is an acronym for service unit.

**Synchronous Checkpoint Update**

When a checkpoint has been created with the synchronous update option, write calls return only when all checkpoint replicas have been updated. In addition, the SA Checkpoint Service guarantees that there are no partial updates in a replica: A replica is either updated with all data specified in the write call or is not updated at all. See also “Asynchronous Checkpoint Update”.

### Switch-Over

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 to another component while the first component is still healthy and capable of providing the service. Switch-over operations are usually the consequence of administrative operations (such as lock of a service unit) or escalation of recovery procedures.

### 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.

### Unicast Message Queue Group

A unicast message queue group is one such that a message is sent to only one message queue in the group, whereas a multicast message queue group is one such that a message can be sent to multiple message queues in the group simultaneously.

### Unlock Administrative Operation

This operation is derived from the ITU X.731 state management model (see [2]). It applies to a service unit, a service group, a cluster node, a service instance, an application, and the cluster.

### Unlocked Service Unit

In the description of the Availability Management Framework, saying that a service unit is unlocked without further specification implies that its administrative state and the administrative states of the following entities is Unlocked: service group containing it, cluster node containing it (for local service units), enclosing application, and cluster (for local service units). If only the administrative state of the service unit is meant, this will be explicitly stated.

### Unlocked State

This is one of the administrative states that are maintained for service units, service groups, nodes, service instances, applications, and the cluster. An entity in this state may be elected to participate in providing service if its operational state is enabled. Refer to “Unlock Administrative Operation” for further details.

### Unregistered Process

Any process of an SA-aware component other than the registered process.

### View Number

A view number is associated with each transition of the cluster membership. For a given view number, all processes obtain the same view of the cluster membership. The view number increases with each membership transition, although not necessarily by one.