NPF HA Service API Implementation Agreement

July 6, 2004

Revision 1.0

Editor(s):

Ram Gopal. L, Nokia, ram.gopal@nokia.com

Santosh Balakrishnan, Intel , santosh.balakrishnan@intel.com

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", “MAY", and "OPTIONAL" in the remainder of this document are to be interpreted as described in the NPF Software API Conventions Implementation Agreement revision 1.0.

For additional information contact:
The Network Processing Forum, 39355 California Street,
Suite 307, Fremont, CA 94538
+1 510 608-5990 phone ✦ info@npforum.org

Table of Contents

1 Revision History

Revision	Date	Reason for Changes
1.0	07/07/2004	Created Rev 1.0 of the implementation agreement

2 Scope and Purpose

This document describes the API definition that will be used by applications that implements HA-SAPI and HA-FAPI [1]. This document in intended for NPF SW HA [1] and HA application implementer. This document identifies SA Forum API's that are relevant for NPF HA implementation. We recommend SA Forum AIS specification [2] for details of each API.

3 Normative References

The following documents contain provisions, which through reference in this text constitute provisions of this specification. At the time of publication, the editions indicated were valid. All referenced documents are subject to revision, and parties to agreements based on this specification are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

1. NPF.2003.404.07 NPF HA architecture model and framework.
2. SAI-AIS-A.01.01 Service Availability Forum Application Interface Specification.
3. NPF.2003.296 NPF HA use case and requirement.
4. Software API Framework Implementation agreement, IA, NPF, Version 1.0
5. NPF.2002.240.27 NPF Packet handler API.
6. NPF.2003.404.01 Proposal for High availability architecture model.
7. Software API Framework Implementation agreement, IA, NPF, Version 1.0,

4 Acronyms and Abbreviations

The following acronyms and abbreviations are used in this specification:

- Card - Line card or control card is referred to a Card.
- Resource - A resource is a logical or physical entity that is managed by HA middleware. Resource may be either HA aware or Non-HA aware. Resource registers with a HA middleware using a component Name.
- HA aware Resource - A HA aware resource is a resource that uses HA middleware APIs and implements functions that need to be invoked by the HA middleware and reports its states and availability information to the other HA aware resources through the HA middleware. Resources cooperate among themselves and provide periodic status to the HA middleware in the form of events. The HA Task Group restricts HA-aware resources to software applications that run on the line and control cards.

  There may be situations, for example, when an HA aware resource first registers with the HA middleware and then forks several child processes (or child resources). In this situation, only the parent process is registered with the HA middleware – i.e. the HA middleware manages the parent process only. . Child processes that need HA services should explicitly register with the HA middleware either under the same component name as parent or may use different component name.
- Non HA aware Resource - Resources that neither register nor use the HA middleware functions and services are classified as non-HA aware resources. A non-HA middleware resource is one that does not provide redundancy. The HA middleware running on line and control card can only perform basic operations like Start and Stop on these resources. The required level of management and monitoring of these resources depends upon the underlying operating system and defining such requirements is beyond the scope of this task group.
- Resource pool - A resource pool is a collection of one or more processes that are registered under the same component name. Resource pool can either reside within a single card or be distributed across several cards.

  A line card or control card is considered as a single unit. One or more applications running under a control or line card may use HA middleware and its services. For example, two control units, say CE1 and CE2, might be running two HA aware applications, say BGP routing daemon and OSPF routing daemon. Each of these HA applications is uniquely identified under a CE and the redundancy is considered across CEs. For instance, CE1 may be in active state and CE2 may be in standby (or hot standby) state. Also, HA resources should be same in both the CEs.
- HA Server (HAS) - Each line card or control card runs an instance of HAS. An HAS is a server that implements the HA API. An HA Server running in line or control cards discovers each other, periodically synchronizes their states and provides a notion of HA middleware to the applications.
- HA-API - The HA framework provides a set of HA APIs to build highly available systems that provide continuous service. It consists of two sets of APIs namely Availability Management Function API(AMF) and Service API (SE). Each HA resource must implement HA AMF API and SE API's.
- HA-FAPI - FAPI implementation that can invoke the HA Service or HA application management API is called HA-FAPI.
- HA-SAPI - SAPI implementation that can invoke HA Service or HA application management API is called HA-SAPI.
- API - Applications Programming Interface
- CE - Control Element also referred as control card
- FAPI - NPF Functional API

- FE - Forwarding Element also referred as line card
- HA - High Availability
- NE - Network Element
- NP - Network Processor
- NPE - Network Processing Element
- NPF - Network Processing Forum
- NPU - Network Processing Unit (same as NPE)
- SAPI - NPF Service API
- FAPI - NPF Functional API
- ForCES - Forwarding and Control Element Separation
- HAS - HA Server
- HA-SAPI - HA aware SAPI
- HA-FAPI - HA aware FAPI
- RHAS - Root HA Server
- BHAS - Backup HA Server
- HAS SET-ID - Unique HA set identifier
- HA-ID - Unique identifier for HAS within a HA SET

5 HA Overview

Highly available systems are designed to protect against network and operational failures. This is usually achieved via redundancy within each network element. Also, network elements are moving from a monolithic software entity to a more distributed function. The high availability functionality should support this distributed architecture. To support high availability in telephony networks, redundancy was built into each network element. However network elements such as routers have evolved from a monolithic software piece to a distributed software and hardware entity. Network elements may have to maintain per-user or aggregate states to satisfy the service requirements of emerging real-time services. Hence, network elements need to provide high-availability features such as fail-over, load-balancing, state replication and resource redundancy in order to avoid disruption in service. This implies that network elements should support high availability features such as fail-over, load-balancing, state replication etc.

This document describes the HA API definition that will be used by HA aware applications that implements HA-SAPI and HA-FAPI. It maps SA Forum API to NPF HA architecture and describes data structure and required data types required to implement the APIs. Finally this document provides implementation guidelines to incorporate HA Service on to existing NPF SW APIs.

6 HA Service API Overview

Registered HA applications are managed by the NPF HA middleware. Figure 1 describes various HA API interfaces. Interface labeled 1 and 2 are internal to HA implementation and are not visible for outside applications. HA application interacts with HA middleware by invoking HA API's. HA API's are labeled as 3 and 3' and are open interfaces. Depending upon the type of NPF SW implementation either the application or the NPF SW API implementation may interact with HA middleware by invoking appropriate HA API functions. HA API includes both Availability management function API (AMF) and Service API. AMF API provides the following services to HA aware applications:

- Registration and deregistration
- Health monitoring
- Availability Management
- Resource pool Management
- Error reporting

The above set of functions are supported by SA Forum APIs in addition to these NPF HA defines additional API that are needed for HA management.

Each NPF HA middleware [1] must implement Event Service and Checkpoint Service. These two services are collectively called as HA Service API. Not all application are required to use the HA Service API. Depending upon the nature of the application one or both or neither of the HA Service may be used by the application. But each application is expected to register and deregister with each HA Service explicitly. Each HA service is independent of each other but they are dependent on AMF API.

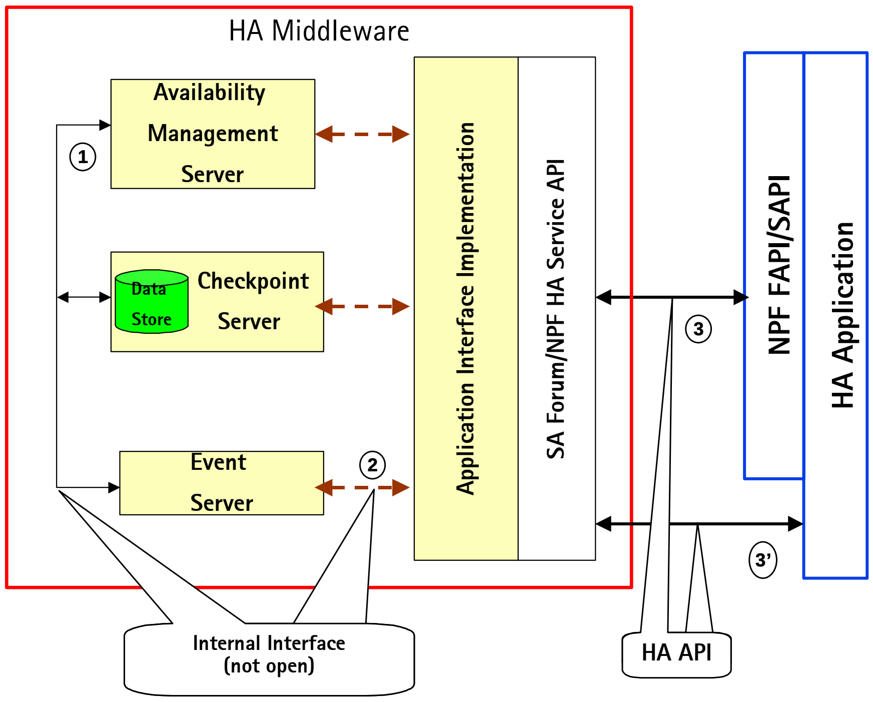

HA Middleware
Availability Management Server
1
Data Store
Checkpoint Server
Event Server
2
Application Interface Implementation
SA Forum/NPF HA Service API
3
3'
NPF FAPI/SAPI
HA Application
Internal Interface (not open)
HA API

Figure 1 NPF HA middleware and HA application interaction

7 Availability Management Framework

The API described in this section only covers interaction and interface required needed between HA aware application and the HA middleware. NPF HA middleware must provide implementation for all these APIs. HA application can invoke these API any given point of time.

7.1 NPF HA extension API

HA-SAPI or HA-FAPI may be implemented as one of the following ways:

- NPF SW API are linked to application either as static or dynamic library
- NPF SW API are integrated as with application (with source code)
- Application may be using one or more HA-SAPI or HA-FAPI libraries (either static or dynamic)
- Application may be using the NPF SW API functions, which are part of the device driver chain (these are specific to certain implementation).

Irrespective of the above cases, each application needs to pass its context to the HA middleware. If the NPF SW API implementation is being shared by several processes [5]then each application needs to be explicitly initialized by the HA middleware and the NPF SW API implementation should not mix the HA context.

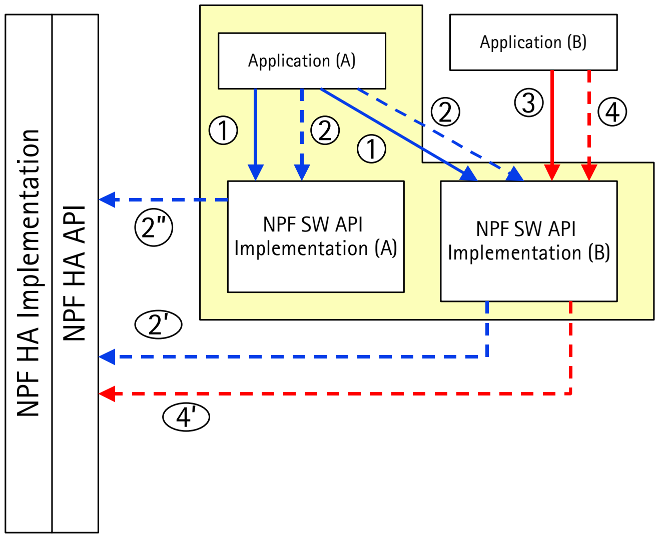


Figure 2 HA application registration and operation sequence

Figure 2 illustrates two HA application namely application A and application B wants to use HA service. Both these application invokes NPF SW API (either HA-FAPI or HA-SAPI or both). For purpose of simplicity we have mentioned as NPF SW API. Application A invokes two NPF SW API may one from vendor “A” and another from vendor “B”. Application B invokes vendor “B” NPF SW APIs. Assume that both NPF SW APIs implementation keep states and needs to synchronize with standby system (not shown in the diagram), it will use checkpoint service. Since the NPF SW API implementation “B” is invoked by both Application A and B, the HA middleware should maintain and manages these states separately. For this purpose each application needs to pass its context during the initial phase of the library initialization (message 1 and 3 from application A and B respectively in Figure 2). It should also be possible for an application to request the HA aware SAPI or FAPI implementation not to perform any HA functions or selectively perform only certain functions. Each SAPI and FAPI must maintain these application specific HA context register on behalf of application. Since application A uses two NPF SW implementation HA middleware will get twice the HA registration message from each NPF SW implementation. Since the registration is for the same application context under it component instance (process ID) it is considered as re-registration and it is perfectly legal.

Following are the list of NPF HA extension API that needs to be supported by HA SAPI and HA-FAPI.

NPF_error_t NPF_XXX_HAInit(…);

/* Initialize the HA context for FAPI or SAPI. This needs to provided by each HA aware SAPI and FAPI */

```
NPF_error_t  NPF_XXX_HADeregister(…);
        /*  De register HA application from the     HA environment */
```

API Data types:

```
        typedef NPF_char8_t    NPF_HA_Component_Name_t;
        typedef NPF_char8_t    NPF_HA_Role;
        typedef  NPF_int32_t   NPF_HA_Correlator_t;
        typedef  NPF_int32_t   NPF_HA_Instance_Id;
```

7.1.1 Application initialization for HA Service

Syntax:

```
NPF_error_t  NPF_HAInit (
        NPF_IN          NPF_HA_Component_Name *component_Name;
        NPF_IN          NPF_HA_Role           role,
        NPF_IN          NPF_HA_Correlator_t   correlator,
        NPF_IN          NPF_HA_Instance_Id    instance_Id,
        NPF_IN          NPF_boolean_t mode,
);
```

Description of function

NPF SW implementation uses checkpoint of event service on behalf of HA application. In order to differentiate different HA applications states, it is required that HA-SAPI and HA-FAPI need to create unique checkpoint name and generate appropriate events. HA application passes HA component name and associated instance information to the HA-SAPI and HA-FAPI.

Input parameters:

Input	**Description**
*component_Name	NULL terminated character string. Unique name of the component. This needs to be standardized for uniform naming. For example: <ServiceName:Protocol:Imp.Specific> Service Name can be routing, Mobility, QoS, Security, Management etc. Protocol can be IPv4, IPv6 or MPLS etc. For example typical routing application that are running in control plane may BGP, OSPF, and RIP etc. To name a BGP server running in a control plane. Routing_BGPv4 For OSPF running in control plane

	Routing_OSPFv3 etc
role	This basically defines the propagation model and failure operation and this is implementation specific. For details how to interpret this field refer implementation guidelines. This field data is transparent to HA-SAPI and HA-FAPI.
instance_Id	32-bity unique run-time identification for the application. It is mainly used to identify multiple instances
correlator	32-bit unique identification for the application in case if it wants to perform state synchronization from the previous HA session.
mode	If this value is TRUE then the HA application wants to run in HA mode and HA-SAPI and HA-FAPI should invoke HA service API calls when needed. If this value is FALSE then the HA application is running in non-HA mode and HA-SAPI or HA-FAPI should not invoke or perform HA function for this application.

Return Code:

- NPF_NO_ERROR: The registration is successful
- NPF_DUPLICATE_INSTANCE: If the HA middleware is configured to run on single instance and if correlator, component_Name and instance are different. Then the HA middleware will generate this error.

7.1.2 De Registration application HA context from HA middleware

Syntax:

```
void   NPF_HADeregister(
            NPF_IN          NPF_HA_Component_Name *component_Name;
            NPF_IN          NPF_HA_Correlator_t   correlator,
            NPF_IN          NPF_HA_Instance_Id    instance_Id,
);
```

Description:

This function is invoked by the application informing the HA-SAPI or HA-FAPI to perform deregistration process from the HA service functions. The HA-SAPI and HA-FAPI flush the internal states if any.

Input parameters:

Input	Description
*component_Name	NULL terminated character string. Unique name of the component. This needs to be standardized for uniform naming. For example: <ServiceName:Protocol:Imp.Specific> Service Name can be routing, Mobility, QoS, Security, Management etc. Protocol can be IPv4, IPv6 or MPLS etc. For example typical routing application that are running in control plane may BGP, OSPF, and RIP etc. To name a BGP server running in a control plane. Routing_BGPv4 For OSPF running in control plane Routing_OSPFv3 etc
instance_Id	32-bity unique run-time identification for the application. It is mainly used to identify multiple instances
correlator	32-bit unique identification for the application in case if it wants to perform state synchronization from the previous HA session.

Return Codes:

- None

7.2 Availability Management Framework API

HA middleware implements following functions as part of availability management framework.

- Registration and deRegistration
- Health Monitoring
- Availability Management
- Protection group management
- Error Reporting

HA aware application must implement call back function for health monitoring, library life cycle management, Protection group and switch over operation.

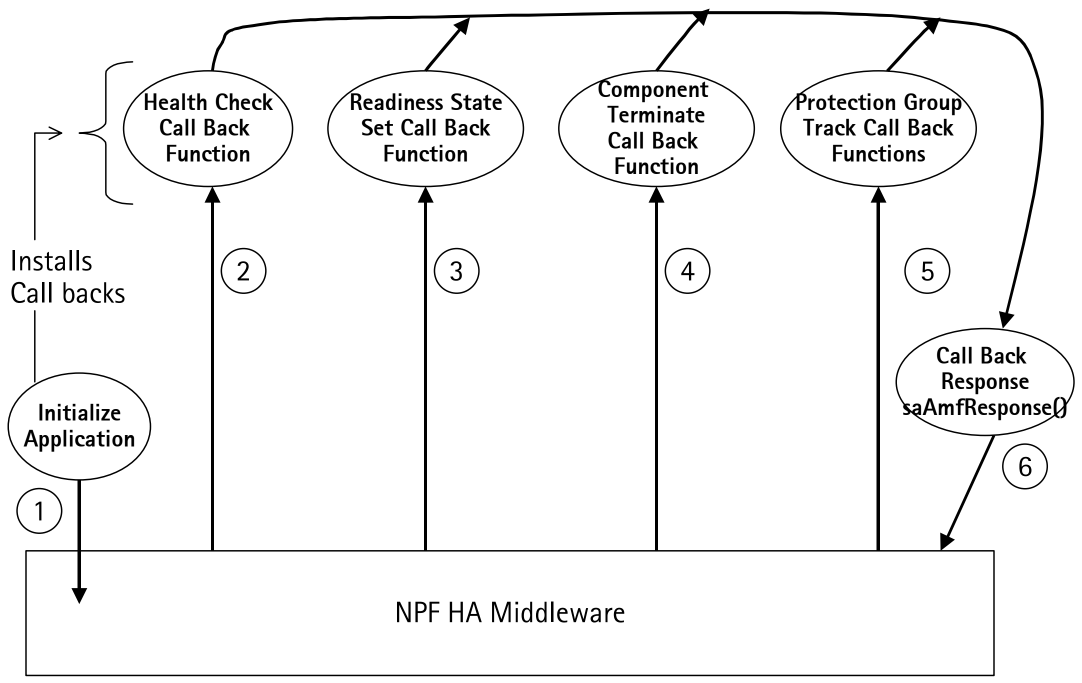

Health Check Call Back Function
Readiness State Set Call Back Function
Component Terminate Call Back Function
Protection Group Track Call Back Functions
Installs Call backs
2
3
4
5
Call Back Response saAmfResponse()
Initialize Application
1
6
NPF HA Middleware

Figure 3 Call back functions for HA middleware management

Figure 3 describes the list call back functions that needs to be implemented by each HA aware process.

- HA application initializes with the HA middleware. This is the first operation that needs be done by any HA aware resource and is described in Figure 3, message 1. During the library initialization, the HA application installs various callbacks for resource management.

- Periodically HA middleware invokes Health check call back functions as described in Figure 3 message 2. HA application generates a response via saAmfResponse() functions.

- When a service unit is available to provide service (that is when a card is inserted onto chassis or the machine is booted up). The HA middleware kicks in processes and starts HA application automatically during boot up process or it can be manually started by operator. When a component is started, the HA middleware assigns the state of the service unit (card) to the HA process. The possible states of card are in-service, out-of-service, stopped. HA middleware at any given time invokes the Readiness state call back function and asking the HA application to changes its readiness state as described Figure 3 message 3. The HA application after making the state transition responds via saAmfResponse() function.

- If HA aware application wants to keep track of the changes to the list of component that are belonging to a given service instance. HA management keeps track of the changes and informs the HA application via protection group track call back function as described in Figure 3 message 5. Though its useful ness is very limited and it's optional to implement this function.

- Components can be terminated at any given time by the HA middleware by invoking the component terminate call back function as described in Figure 3 message 4. Note component termination simply deregisters the component from HA middleware, HA application needs to be perform additional clean up functions to disassociate completely from the HA middleware.

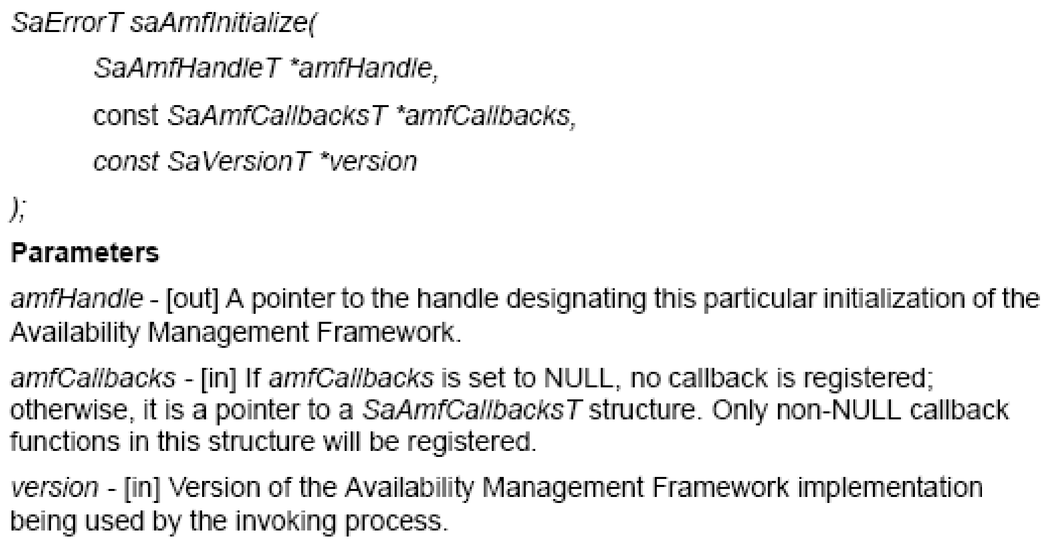
- All the callback function respond to the HA middleware queries by invoking the saAMFresponse function. Each call back function contain an invocation reference which will be passed back along with the response to the HA middleware. This is illustrated in message 6 in Figure 3.

The following are the lists of SA Forum API are be applicable to NPF HA environment. For detailed description of functions and the parameters refer SA Forum AIS specification [2].

7.2.1 Library Lifecycle

All AMF library lifecycle API will be invoked by HA application.

Syntax:



```
SaErrorT saAmfInitialize(
        SaAmfHandleT *amfHandle,
        const SaAmfCallbacksT *amfCallbacks,
        const SaVersionT *version
);
```

Parameters

amfHandle - [out] A pointer to the handle designating this particular initialization of the Availability Management Framework.

amfCallbacks - [in] If *amfCallbacks* is set to NULL, no callback is registered; otherwise, it is a pointer to a *SaAmfCallbacksT* structure. Only non-NULL callback functions in this structure will be registered.

version - [in] Version of the Availability Management Framework implementation being used by the invoking process.

Description:

This is the first API call that HA aware application must make to HA middleware. HA application initializes various callbacks which will be invoked by HA middleware.

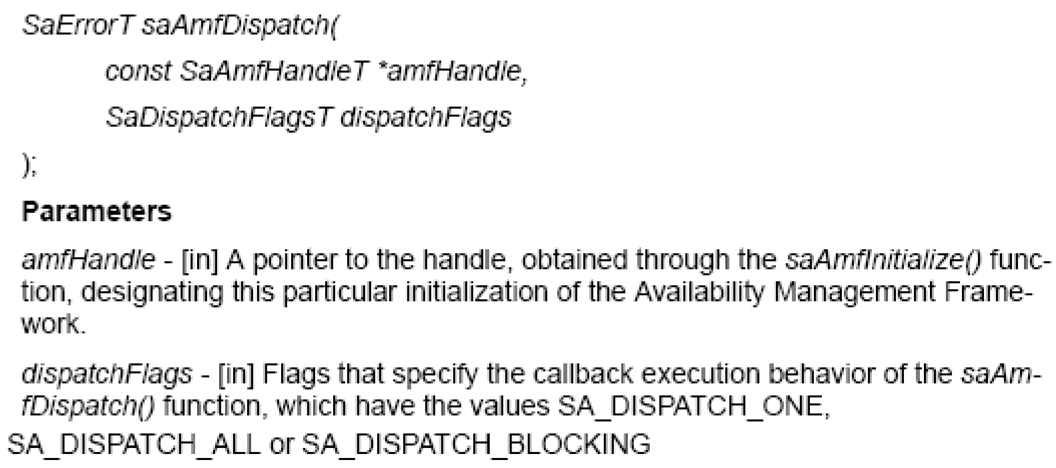
The following are the minimum call back functions that are required in NPF HA environment.

```
typedef struct {
   SaAmfHealthcheckCallbackT
      saAmfHealthcheckCallback;              //Required
```

```
    SaAmfReadinessStateSetCallbackT
      saAmfReadinessStateSetCallback;  // Required
    SaAmfComponentTerminateCallbackT
                saAmfComponentTerminateCallback;    // Required
    SaAmfCSISetCallbackT
      saAmfCSISetCallback;              //Required
    SaAmfCSIRemoveCallbackT
                saAmfCSIRemoveCallback;             //Required
    SaAmfProtectionGroupTrackCallbackT
      saAmfProtectionGroupTrackCallback;      //Optional
    SaAmfExternalComponentRestartCallbackT
                saAmfExternalComponentRestartCallback;      //Set to NULL
    SaAmfExternalComponentControlCallbackT
                saAmfExternalComponentControlCallback;      //Set to NULL
    SaAmfPendingOperationConfirmCallbackT
      saAmfPendingOperationConfirmCallback;         //Set to NULL
  } SaAmfCallbacksT;
```

7.2.2 Calling back Sequence

Syntax:



```
SaErrorT saAmfDispatch(
        const SaAmfHandleT *amfHandle,
        SaDispatchFlagsT dispatchFlags
);
```

Parameters

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

dispatchFlags - [in] Flags that specify the callback execution behavior of the *saAmfDispatch()* function, which have the values SA_DISPATCH_ONE, SA_DISPATCH_ALL or SA_DISPATCH_BLOCKING

Description:

HA application can specify the semantics for call back whether the HA middleware can perform blocking call (SA_DISPATCH_BLOCKING) or dispatch all (SA_DISPATCH_ALL) call back at once or dispatch one (SA_DISPATCH_ONE) call back at a time.

We recommend SA_DISPATCH_ONE to be used if the HA aware is a single threaded process, and if the use of SA_DISPATCH_BLOCKING can be used if the HA aware process is performs one operations at a time. This is totally application specific.

7.2.3 De-registering the HA application from HA middleware

Syntax:

```
SaErrorT saAmfFinalize(
        const SaAmfHandleT *amfHandle
);
```

Parameters

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

Description:

This function will be invoked by the HA application to de register completely from the HA middleware. This function call releases all the resources from the HA middleware.

7.2.4 Component registration

Syntax:

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

Parameters

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

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

ProxyCompName is set to NULL in NPF HA environment.

Description:

After initializing the callback functions, the HA application needs to perform component registration with the HA middleware. Component name is unique and is used to identify the HA resource (in our NPF HA it is referred to application process). In NPF we don't have the notion of proxy component and hence its value is set to NULL.

7.2.5 Component Deregistration

Syntax:

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

Parameters

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

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

proxyCompName is set to NULL in NPF HA environment.

Description:

Component can deregister at any given time by invoking this API function. After de registering the component the HA aware application should not process any network packets. HA aware application can perform any number component registration and deregistration any number of times. If a component is deregistered, then the HA middleware can perform switch over operation if continuous service needs to be provided for that component.

7.2.6 Health check request

Syntax:

```
typedef void (*SaAmfHealthcheckCallbackT)(
        SaInvocationT invocation,
        const SaNameT *compName,
        SaAmfHealthcheckT checkType
);
```

Parameters

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

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

checkType - [in] The type of the healthcheck to be executed.

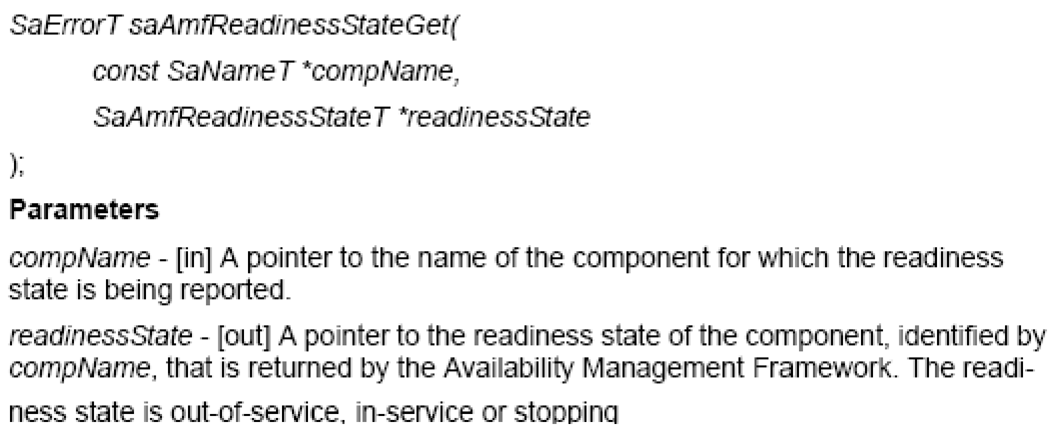
In NPF the checkType simple liveness is recommended. The Value of checktype is set to SA_AMF_HEARTBEAT =1. Other checktypes are beyond the current scope of HA middleware.

Description:

HA middleware invokes the health check call back functions and asking the HA application to respond to with its internal state. HA application must respond to this request by invoking saAmfResponse() function with appropriate invocation value.

7.2.7 Get Component status

Syntax:



```
SaErrorT saAmfReadinessStateGet(
        const SaNameT *compName,
        SaAmfReadinessStateT *readinessState
);
```

Parameters

compName - [in] A pointer to the name of the component for which the readiness state is being reported.

readinessState - [out] A pointer to the readiness state of the component, identified by *compName*, that is returned by the Availability Management Framework. The readi-

ness state is out-of-service, in-service or stopping

Description:

It there are dependency between two HA aware control plane applications, one HA application after registering with the HA middleware can make queries to the HA framework to get to know the status of the other HA aware application by specifying the component name. The HA middleware will report the readiness status of the other application. This API can be used to determine and control the behavior of application process. Usage of this API is application dependent.

7.2.8 Change the Readiness state of the Application

Syntax:

```
typedef void (*SaAmfReadinessStateSetCallbackT)(
        SaInvocationT invocation,
        const SaNameT *compName,
        SaAmfReadinessStateT readinessState
);
```

Parameters

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

compName - [in] A pointer to the name of the component whose readiness state the Availability Management Framework is setting.

readinessState - [in] The readiness state of the component, identified by *compName*, that is being set by the Availability Management Framework.

Description:

At any given time HA middleware request the HA aware application to change its readiness states. This transition may be due to operator changing the service unit (card) state to either out-of-service or stopped.

7.2.9 Terminate a HA component

Syntax:

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

Parameters

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

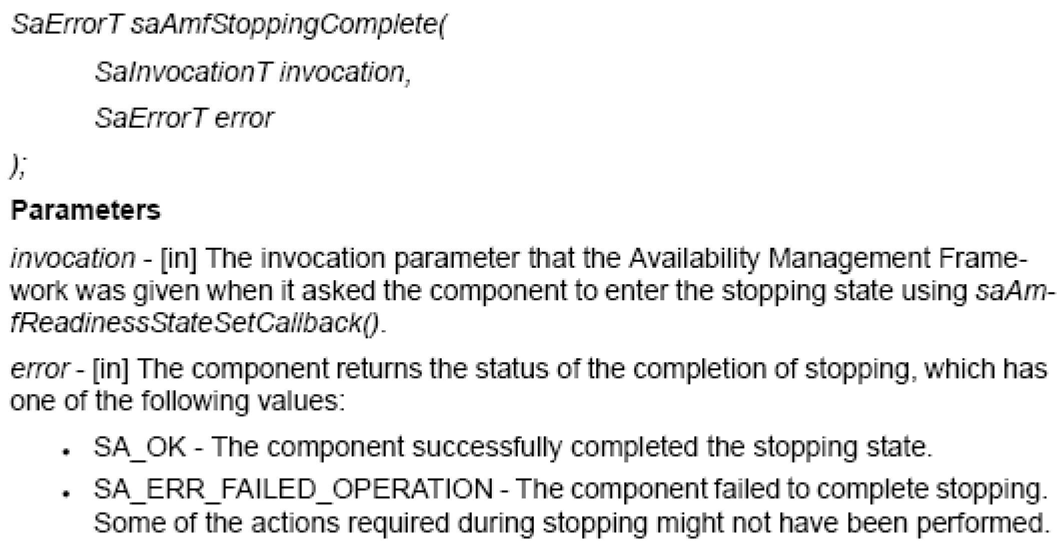
compName - [in] A pointer to the name of the component to be terminated.

Description:

HA middleware sends termination command to HA application and requesting it to close all the operation and disassociate the component from the HA middleware.

7.2.10 Component termination response to HA Middleware

Syntax:



```
SaErrorT saAmfStoppingComplete(
        SaInvocationT invocation,
        SaErrorT error
);
```

Parameters

invocation - [in] The invocation parameter that the Availability Management Frame-work was given when it asked the component to enter the stopping state using *saAmfReadinessStateSetCallback()*.

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

- SA_OK - The component successfully completed the stopping state.
- SA_ERR_FAILED_OPERATION - The component failed to complete stopping. Some of the actions required during stopping might not have been performed.

Description:

This is a response generated by HA application to provide its termination status to the HA middleware. When the HA middleware wants to terminate a HA application (see section 7.2.9) it invokes the HA application terminate call back function. HA application performs clean up operations and will not service any request and sends the termination command via this function. In NPF we have each HA application process to be associated with the component name, terminating the component should stop packet processing and the HA aware application should release all resource. It should not terminate the process.

7.2.11 Set HA state for a component

Syntax:

```
typedef void (*SaAmfCSISetCallbackT)(
        SaInvocationT invocation,
        const SaNameT *compName,
        const SaNameT *csiName,
        SaAmfCSIFlagsT csiFlags,
        SaAmfHAStateT *haState,
        SaNameT *activeCompName,
        SaAmfCSITransitionDescriptorT transitionDescriptor
);
```

Parameters

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

compName - [in] A pointer to the name of the component to which a new component service instance is added or for which the HA state of one or all supported component service instances is changed.

csiName - [in] A pointer to the name of a new component service instance to be supported by the component or of an already supported component service instance whose HA state is to be changed. If the state of all component service instances is to be changed (if SA_AMF_CSI_ALL_INSTANCES is set in *csiFlags*), *csiName* is not meaningful and is set to NULL.

csiFlags - [in] A value of the *SaAmfCSIChoiceT* flags type which indicates whether the HA state change must be applied to a new component service instance or to all component service instances currently supported by the component. If no flags are set, the HA state change applies to a component service instance already supported by the component.

haState - [in] The new HA state to be assumed by the component service instance, identified by *csiName*, or by all component service instances already supported by the component (if SA_AMF_CSI_ALL_INSTANCES is set in *csiFlags*).

activeCompName - [in] A pointer to the name of the component that currently has the active state or had the active state for this component service instance previously. The semantics attached to this parameter varies with the particular HA state being assigned.

transitionDescription - [in] This parameter is meaningful only when *haState* is set to SA_AMF_ACTIVE, in which case it indicates whether or not the component service instance for *activeCompName* went through quiescing, as defined by the *SaAmfCSI-TransitionDescriptorT*

Description:

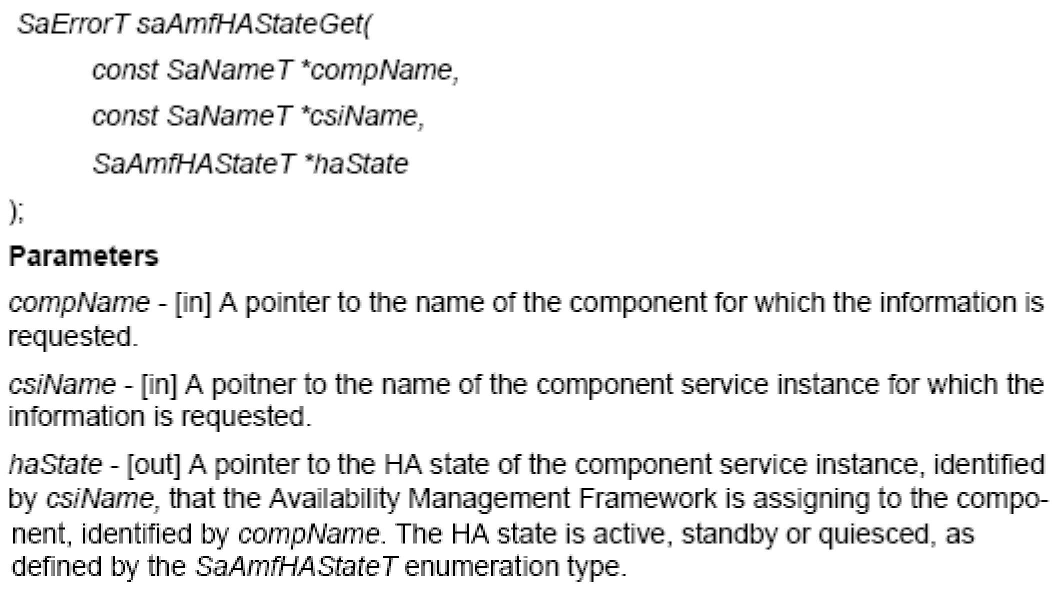
HA states can be classified into two-stage operation. HA aware application first needs to be in-service (meaning that it can be invoked and is executing) but that does not convey whether it can process packet or do normal application functions. In order to control that behavior when the HA application is in-service it can be either be active or standby or quiesced. HA middleware will reflect the state of service unit (card) for HA resources contained in that service unit (card). In NPF component name and component instance name is same, we manage one HA application process at a time.

Following are the parameters in NPF HA environment.

- compName, activeCompName and CSIName all refers to same name. In NPF we are providing 1:1 redundancy model. Each component needs to register explicitly with the HA middleware.
- csiFlag should be set to SA_AMF_CSI_ALL_INSTANCES
- haState can be either ACTIVE,STANBY, QUIESCED
- transitionDescription not used in NPF set to NULL.

7.2.12 Get HA State of a component

Syntax:



```
SaErrorT saAmfHAStateGet(
        const SaNameT *compName,
        const SaNameT *csiName,
        SaAmfHAStateT *haState
);
```

Parameters

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

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

haState - [out] A pointer to the HA state of the component service instance, identified by *csiName*, that the Availability Management Framework is assigning to the component, identified by *compName*. The HA state is active, standby or quiesced, as defined by the *SaAmfHAStateT* enumeration type.

Description:

If a HA application wants to know about the status of itself or other HA aware component running under HA middleware. It can invoke this function to get the status of the other component. For example, if one HA aware component wants to know the status of the other HA aware component, we recommend that first that say HA aware application A needs to determine whether the HA aware application B’s readiness and then only it should invoke this call.

7.2.13 Confirming HA state before performing fail-over or shutdown operation.

Syntax:

```
typedef void (*SaAmfPendingOperationConfirmCallbackT)(
        const SaInvocationT invocation,
        const SaNameT *compName,
        SaAmfPendingOperationFlagsT pendingOperationFlags
);
```

Parameters

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

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

pendingOperationFlags - [in] The operations for which confirmation is requested. This parameter is of the type *SaAmfPendingOperationFlagsT*

which has the values *SA_AMF_SWITCHOVER_OPERATION* and *SA_AMF_SHUTDOWN_OPERATION* or both.

Description:

HA aware application may be processing packets and heavily loaded. Before performing shutdown or switch over operation, the HA middleware request the HA aware application and asking whether it can now perform those function or it should defer later. This is mainly to achieve graceful shutdown and enable seamless load transfer during the fail or shutdown operation. The HA aware application sends its response via saAmfResponse function.

7.2.14 Cancel pending operation

Syntax:

```
typedef void (*SaAmfPendingOperationExpiredCallbackT)(
        const SaNameT *compName,
        SaAmfPendingOperationFlagsT pendingOperationFlags
);
```

Parameters

compName - [in] A pointer to the name of the component on which the pending operation(s) confirmation is being cancelled.

pendingOperationFlags - [in] The operation(s) that have been cancelled.

Description:

HA middleware can inform the HA application to cancel any pending request that it had previously invoked via pending operation confirm call back (see section 7.2.13).

7.2.15 HA application response to HA middleware queries.

Syntax:

```
SaErrorT saAmfResponse(
        SaInvocationT invocation,
        SaErrorT error
);
```

Parameters

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

error - [in] The response of the process to the associated callback. It returns SA_OK if the associated callback is successfully executed by the process. Otherwise, it returns an appropriate error as described in the corresponding callback.

Description:

HA middleware queries HA application via callbacks. The HA application reacts to the call back and provide response via this function. The main reason to have indirect response is that call back can be made either blocking or non-blocking. HA application needs to copy the invocation value which was supplied during the callback invocation by HA middleware in this function.

7.3 Check point Service API

HA application can optionally use HA Checkpoint Service API. But HA middleware must implement Checkpoint service as part of HA implementation. Checkpoint service provides following functions:

- Checkpoint implementation stores the critical data either in the main memory or in the secondary store. For performance reasons usually checkpoint store is in main memory. It replicates the content in more than one places, in order to protect the data against card failures.
- If an HA application crashes the Checkpoint service provides mechanism to resynchronize the state of the application from the previous session. Note such use of synchronization is time and application dependent. It may happen that such resynchronization may be stale data and use of such feature is totally application dependent.

7.3.1 Usage Model and HA implementation guidelines

- HA application (process) can use several checkpoints to save the state of the process. Each checkpoint is identified by a unique name in a HA middleware. HA middleware treats those checkpoint as opaque data store. It is totally up to the implementation to interpret the content, and also when to checkpoint the data or replicate the data.

- Multiple processes can open the same checkpoint and writes to the same checkpoint. It is the responsibility of the process to cooperate among them when they use the checkpoint store. For example, if two telnet servers have forked several processes and they would like to checkpoint some data under a same name. In this scenario, the either they can write to the same memory (over write) or append to the existing checkpoint data. Another example is that two HA aware resources (two processes running in control card) performs some IPC and saves state under the same checkpoint name.

- When an HA aware resources deletes the checkpoint the content is deleted and resources are deleted. But when more than one HA resources opened the same checkpoint, the HA middleware implementation must keep track of the reference count mechanism and it should delete the checkpoint only when the last process has deleted the checkpoint.

- Checkpoint store has some retention time, in order to avoid memory overrun. When HA process terminates, the HA middleware must hold the checkpoint store for some duration of time. After the checkpoint retention timer is expired, the HA middleware should perform cleanup operation.

- HA aware resource would like to perform partial updates rather than full update to whole checkpoint store. To enable this operation, the checkpoint store can be subdivided in to sections. Section is portions of the checkpoint store allocated to HA aware resources. HA resources needs to create sections, and can reference each section under a unique name. HA aware resource needs to specify the number of section, section size and other parameter during the initialization. Each section have different lifetime and is different from checkpoint lifetime.

- Checkpoint store management is part of HA middleware management and is transparent to the HA application. In NPF it is recommended that checkpoint store be managed with the card where the local resource are using it. This will reduce the inter card communication between the checkpoint store and the actual HA resource running in the card. Checkpoint can store many copies of the same data the storage and retrieval of information and how many copies depends upon the type of HA application and architecture. At least we recommend that active card can have one checkpoint store and standby card to have another checkpoint store.

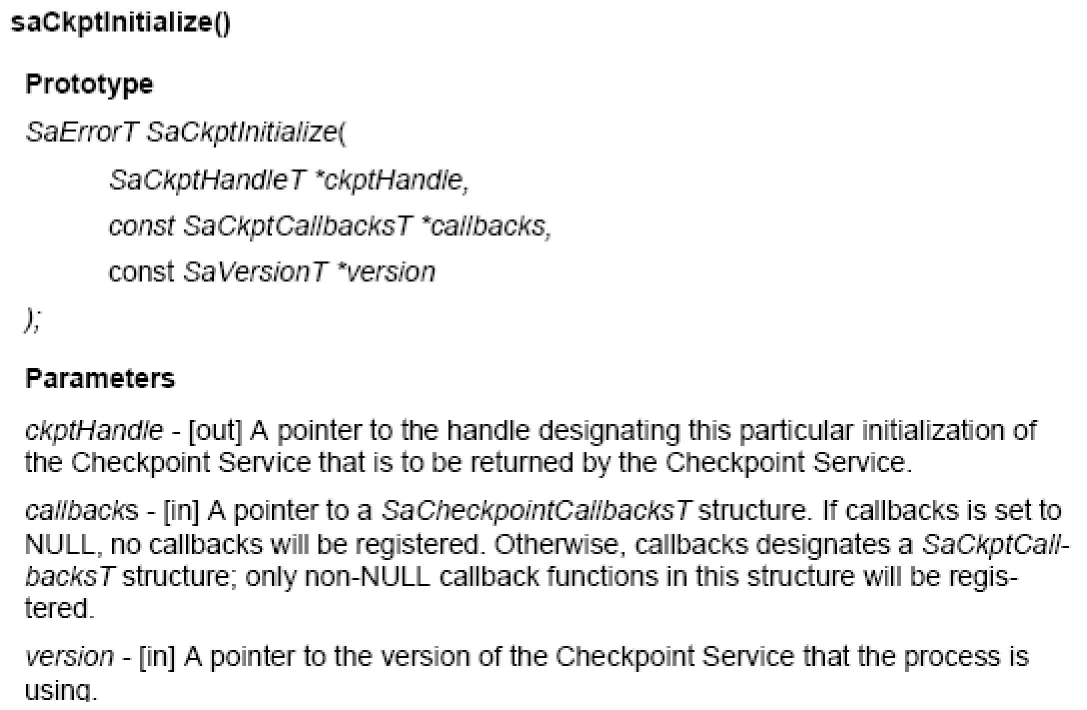
- Operations on checkpoint, for example two application can write to the same checkpoint and the order of write needs to be cooperatively ensure by the application if its required (see use of Event Service) or by the HA middleware implementation. It is up to the HA implementation and is implementation specific.

- HA implementation must support synchronous write and asynchronous write to checkpoint store. When a application writes using the synchronous write call, the checkpoint makes copies of the same data to all the replicas and then return to the HA application. Where as in asynchronous call, the checkpoint store immediately return to the application and at later time, the checkpoint implementation updates those information to the replicas.

- Though the checkpoint application service depends upon the cluster service of HA forum. We are dealing with checkpoint across CE and FE systems hence we narrow the scope of cluster to service group, which are running similar applications.

7.3.2 Checkpoint Library initialization

Syntax:



saCkptInitialize()

Prototype

```
SaErrorT SaCkptInitialize(
        SaCkptHandleT *ckptHandle,
        const SaCkptCallbacksT *callbacks,
        const SaVersionT *version
);
```

Parameters

ckptHandle - [out] A pointer to the handle designating this particular initialization of the Checkpoint Service that is to be returned by the Checkpoint Service.

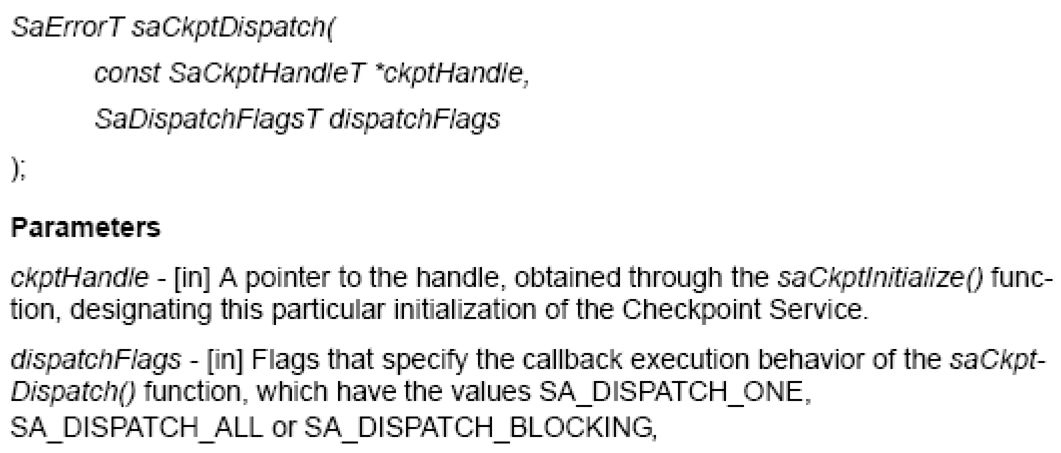
callbacks - [in] A pointer to a *SaCheckpointCallbacksT* structure. If callbacks is set to NULL, no callbacks will be registered. Otherwise, callbacks designates a *SaCkptCallbacksT* structure; only non-NULL callback functions in this structure will be registered.

version - [in] A pointer to the version of the Checkpoint Service that the process is using.

Description:
Each HA aware resource needs to initialize the checkpoint library in order to make use of the checkpoint service. In NPF either if HA-SAPI or HA-FAPI manages state information, then it will invoke this call and will installs appropriate callback functions. If both HA aware resource and HA-SAPI or HA-FAPI are required to use the checkpoint, then HA aware resource should initialize the library. HA implementation must return a unique handle each time when this checkpoint service is being initialized.

7.3.3 Installing dispatch mechanism

Syntax:



```
SaErrorT saCkptDispatch(
        const SaCkptHandleT *ckptHandle,
        SaDispatchFlagsT dispatchFlags
);
```

Parameters

ckptHandle - [in] A pointer to the handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

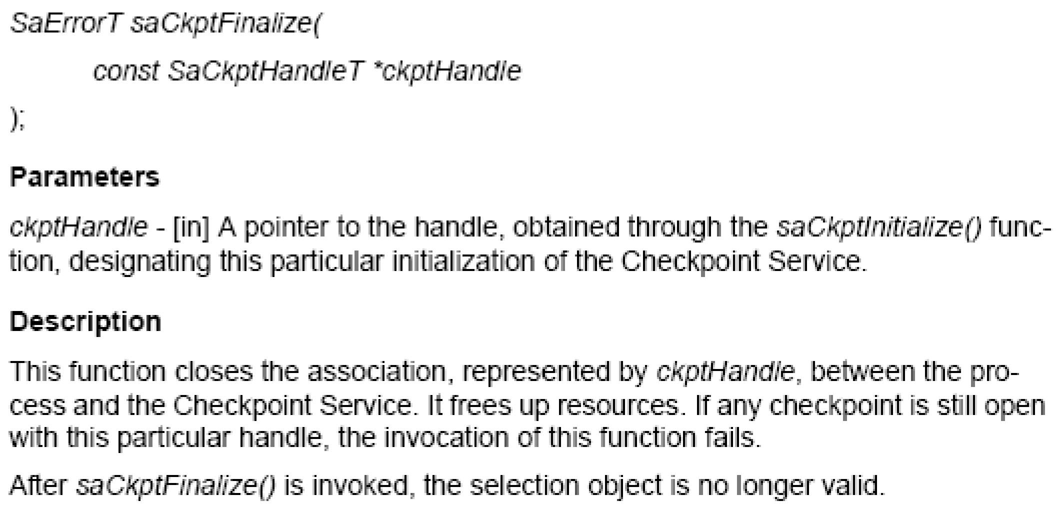
dispatchFlags - [in] Flags that specify the callback execution behavior of the *saCkptDispatch()* function, which have the values SA_DISPATCH_ONE, SA_DISPATCH_ALL or SA_DISPATCH_BLOCKING,

Description:

This function needs to be invoked by the HA aware resource in order to inform the type of dispatch. HA middleware may invoke one call back at a time, or it will dispatch all the pending activities in one call back or it will in invoke the callback in blocking mode. This is implementation dependent, we expect that at least SA_DISPATCH_BLOCKING needs to be supported in all HA middleware implementation.

7.3.4 Detaching the HA aware resource from Checkpoint Service

Syntax:



```
SaErrorT saCkptFinalize(
        const SaCkptHandleT *ckptHandle
);
```

Parameters

ckptHandle - [in] A pointer to the handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

Description

This function closes the association, represented by *ckptHandle*, between the process and the Checkpoint Service. It frees up resources. If any checkpoint is still open with this particular handle, the invocation of this function fails.

After *saCkptFinalize()* is invoked, the selection object is no longer valid.

7.3.5 Opening a checkpoint store (Sync and Async open)

Syntax:

saCkptCheckpointOpen() and saCkptCheckpointOpenAsync()

Prototype

```
SaErrorT saCkptCheckpointOpen(
        const SaNameT *ckeckpointName,
        const SaCkptCheckpointCreationAttributesT *checkpointCreationAttributes,
        SaCkptCheckpointOpenFlagsT checkpointOpenFlags,
        SaTimeT timeout,
        SaCkptCheckpointHandleT *checkpointHandle
);

SaErrorT saCkptCheckpointOpenAsync(
        const SaCkptHandleT *ckptHandle,
        SaInvocationT invocation,
        const SaNameT *ckeckpointName,
        const SaCkptCheckpointCreationAttributesT *checkpointCreationAttributes,
        SaCkptCheckpointOpenFlagsT checkpointOpenFlags
);
```

Parameters

ckptHandle - [in]A pointer to the handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

invocation - [in] A parameter designates a particular invocation of the response callback.

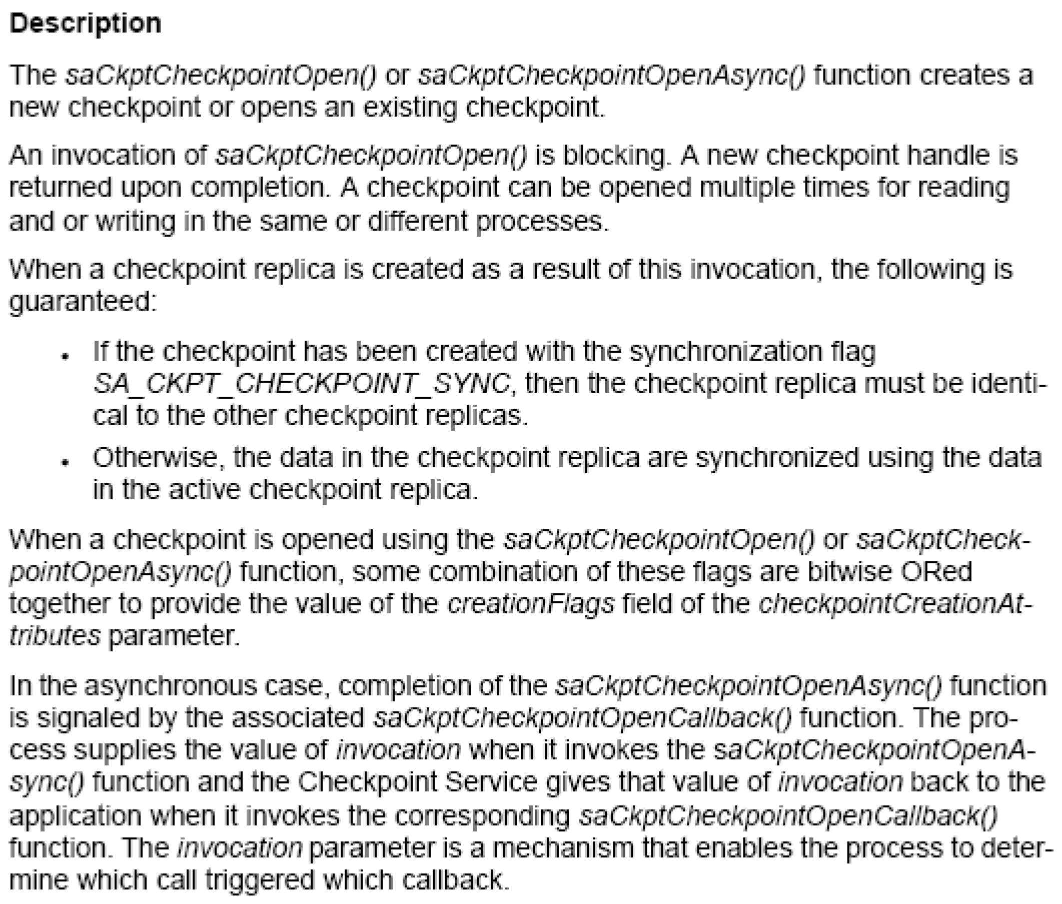
ckeckpointName - [in] A pointer to the name of the checkpoint that identifies a checkpoint globally in a cluster.

checkpointCreationAttributes - [in] A pointer to the creation attributes of a checkpoint. These attributes are relevant only when a checkpoint is created. It is an error to open a checkpoint with creation attributes different from the ones used at creation time. If the intent is only to open a checkpoint, these attributes shall be set to NULL. Otherwise, if the intent is to create a new checkpoint, the *SaCkptCheckpointCreationAttributesT* structure shall contain the attributes for the checkpoint

checkpointOpenFlags - [in] The value of this parameter is constructed by a bitwise OR of the flags defined by the *SaCkptCheckpointOpenFlagsT* type

timeout - [in] The *saCkptCheckpointOpen()* invocation is considered to have failed if it does not complete by the time specified. A checkpoint replica may still be created.

checkpointHandle - [out] A pointer to the checkpoint handle, stored in the address space of the invoking process, that is used to access the checkpoint in subsequent invocations of the functions of the Checkpoint Service API. In the case of *saCkptCheckpointOpenAsync()*, this handle is returned in the corresponding callback.



Description

The *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* function creates a new checkpoint or opens an existing checkpoint.

An invocation of *saCkptCheckpointOpen()* is blocking. A new checkpoint handle is returned upon completion. A checkpoint can be opened multiple times for reading and or writing in the same or different processes.

When a checkpoint replica is created as a result of this invocation, the following is guaranteed:

- If the checkpoint has been created with the synchronization flag *SA_CKPT_CHECKPOINT_SYNC*, then the checkpoint replica must be identical to the other checkpoint replicas.
- Otherwise, the data in the checkpoint replica are synchronized using the data in the active checkpoint replica.

When a checkpoint is opened using the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* function, some combination of these flags are bitwise ORed together to provide the value of the *creationFlags* field of the *checkpointCreationAttributes* parameter.

In the asynchronous case, completion of the *saCkptCheckpointOpenAsync()* function is signaled by the associated *saCkptCheckpointOpenCallback()* function. The process supplies the value of *invocation* when it invokes the *saCkptCheckpointOpenAsync()* function and the Checkpoint Service gives that value of *invocation* back to the application when it invokes the corresponding *saCkptCheckpointOpenCallback()* function. The *invocation* parameter is a mechanism that enables the process to determine which call triggered which callback.

7.3.6 Checkpoint Open Call back

Syntax:

```
typedef void (*SaCkptCheckpointOpenCallbackT)(
        SaInvocationT invocation,
        const SaCkptCheckpointHandleT *checkpointHandle,
        SaErrorT error
);
```

Parameters

invocation - [in] This parameter was supplied by a process in the corresponding invocation of the *saCkptCheckpointOpenAsync()* function and is used by the Checkpoint Service in this callback. This invocation parameter allows the process to match the invocation of that function with this callback.

checkpointHandle - [in] A pointer to the handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

error - [in] The error returned if the *saCkptCheckpointOpenAsync()* function was not successful. The error codes that can be returned are as follows:

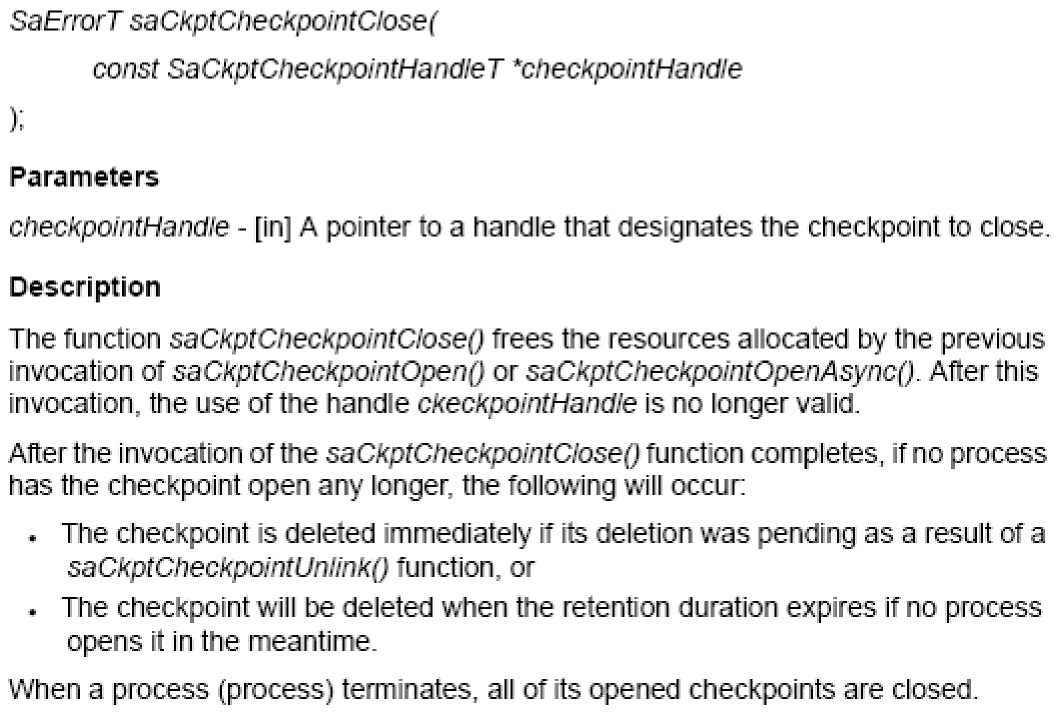
- SA_OK - The function completed successfully.
- SA_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.
- SA_ERR_INIT - The function *saCkptInitialize()* has not been invoked yet or the function *saCkptFinalize()* has already been invoked.
- SA_ERR_TIMEOUT - An implementation-dependent timeout occurred or the timeout defined by the *timeout* parameter occurred before the call could complete. It is unspecified whether or not the call succeeded.
- SA_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may try again.
- SA_ERR_INVALID_PARAM - A parameter is not correctly set.
- SA_ERR_NO_MEMORY - Either the local library or a process that is providing the service is out of memory, and cannot provide the service.
- SA_ERR_ACCESS - The checkpoint exists but the permissions specified by the *checkpointOpenFlags* parameter are denied.
- SA_ERR_NOT_EXIST - The *creationAttributes* parameter is NULL and the checkpoint does not exist.
- SA_ERR_EXIST - The checkpoint already exists and the *creationAttributes* are different from the ones used at creation time.
- SA_ERR_BAD_FLAGS - The *checkpointOpenFlags* parameter is invalid.

Description

The Checkpoint Service invokes this callback function when the invocation of *saCkptCheckpointOpenAsync()* is completed. If successful, the reference to the opened/created checkpoint is returned in *checkpointHandle;* otherwise, an error is returned in the error parameter.

7.3.7 Close the checkpoint

Syntax:



```
SaErrorT saCkptCheckpointClose(
        const SaCkptCheckpointHandleT *checkpointHandle
);
```

Parameters

checkpointHandle - [in] A pointer to a handle that designates the checkpoint to close.

Description

The function *saCkptCheckpointClose()* frees the resources allocated by the previous invocation of *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()*. After this invocation, the use of the handle *ckeckpointHandle* is no longer valid.

After the invocation of the *saCkptCheckpointClose()* function completes, if no process has the checkpoint open any longer, the following will occur:

- The checkpoint is deleted immediately if its deletion was pending as a result of a *saCkptCheckpointUnlink()* function, or
- The checkpoint will be deleted when the retention duration expires if no process opens it in the meantime.

When a process (process) terminates, all of its opened checkpoints are closed.

HA-SPI or HA-FAPI which is being used by multiple HA aware resource should perform clean up operation, when the HA application invokes the HA extension API to deregister from the service.

7.3.8 Set checkpoint retention time

Syntax:

```
SaErrorT saCkptCheckpointRetentionDurationSet(
        const SaCkptCheckpointHandleT *checkpointHandle,
        SaTimeT retentionDuration
);
```

Parameters

checkpointHandle - [in] A pointer to the checkpoint whose retention time is being set.

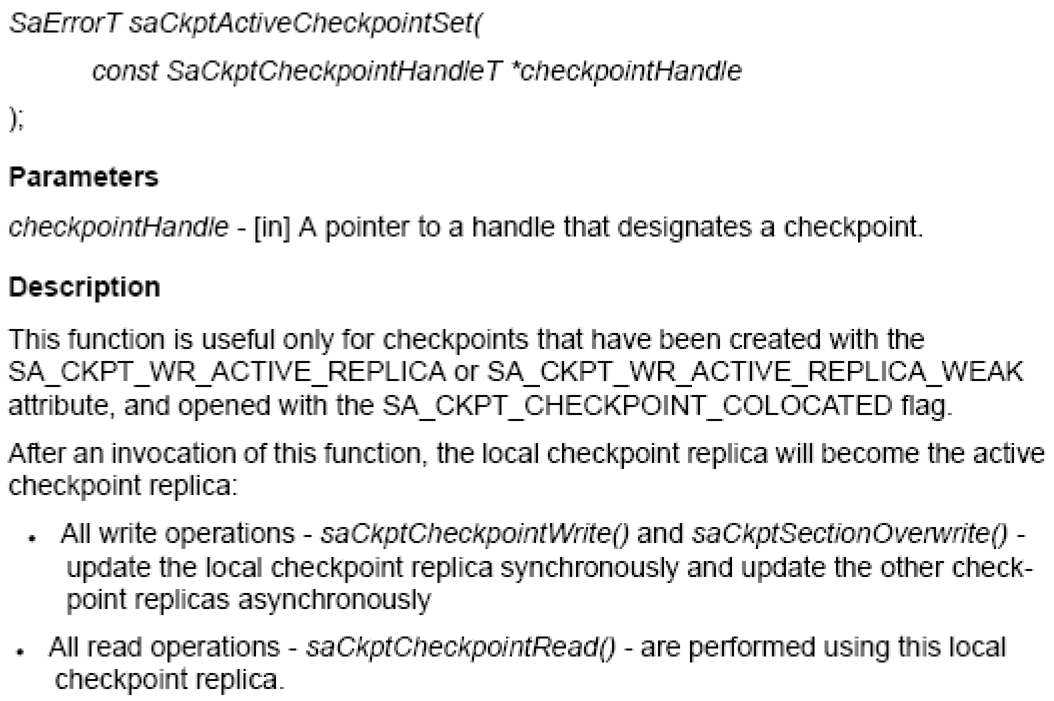
retentionDuration - [in] The value of the retention duration to be set. The checkpoint is retained (not deleted) for the retention duration.

Description

The function *saCkptCheckpointRetentionDurationSet()* sets the retention duration of the checkpoint, designated by *checkpointHandle,* to *retentionDuration*. After no more processes have the checkpoint open and if the checkpoint is not opened by any process for the retention duration, the Checkpoint Service automatically deletes the checkpoint.

7.3.9 Set active checkpoint to active replica

Syntax:



```
SaErrorT saCkptActiveCheckpointSet(
        const SaCkptCheckpointHandleT *checkpointHandle
);
```

Parameters

checkpointHandle - [in] A pointer to a handle that designates a checkpoint.

Description

This function is useful only for checkpoints that have been created with the SA_CKPT_WR_ACTIVE_REPLICA or SA_CKPT_WR_ACTIVE_REPLICA_WEAK attribute, and opened with the SA_CKPT_CHECKPOINT_COLOCATED flag.

After an invocation of this function, the local checkpoint replica will become the active checkpoint replica:

- All write operations - *saCkptCheckpointWrite()* and *saCkptSectionOverwrite()* - update the local checkpoint replica synchronously and update the other checkpoint replicas asynchronously
- All read operations - *saCkptCheckpointRead()* - are performed using this local checkpoint replica.

7.3.10 Get the checkpoint status

Syntax:

```
SaErrorT saCkptCheckpointStatusGet(
        const SaCkptCheckpointHandleT *checkpointHandle,
        SaCkptCheckpointStatusT *checkpointStatus
);
```

Parameters

checkpointHandle - [in] A pointer to a handle of the checkpoint whose status is to be returned

checkpointStatus -[out] A pointer to a *SaCkptCheckpointStatusT* structure

Description

This function retrieves the *checkpointStatus* of the checkpoint designated by *check-pointHandle*.

7.3.11 Create checkpoint section

Syntax:

```
SaErrorT saCkptSectionCreate(
        const SaCkptCheckpointHandleT *checkpointHandle,
        SaCkptSectionCreationAttributesT *sectionCreationAttributes,
        const void *initialData,
        SaUint32T initialDataSize
);
```

Parameters

checkpointHandle - [in] A pointer to the handle of the checkpoint that is to hold the section. The checkpoint handle *checkpointHandle* must be obtained by a previous

invocation of the *saCkptCheckpointOpen() function* with the SA_CKPT_CHECKPOINT_WRITE flag set.

sectionCreationAttributes - [in] A pointer to a structure *SaCkptSectionCreationAt-tributesT*, that contains the in/out field *sectionId* and the

initialData - [in] A location in the address space of the invoking process that contains the initial data of the section to be created.

initialDataSize - [in] The size of the initial data of the section to be created. Initial size can be at most *maxSectionSize,* as specified by the checkpoint creation attributes in *saCkptCheckpointOpen()*.

Description

This function creates a new section. Unlike a checkpoint, a section does not need to be opened for access. The section shall be deleted by the Checkpoint Service when its expiration time is reached. If a checkpoint is created to have only one section, it is not necessary to create that section. The default section is identified by the special identifier SA_CKPT_DEFAULT_SECTION_ID. If the SA_CKPT_WR_ALL_REPLICAS property is set, the section is created in all of the checkpoint replicas when the invocation returns; otherwise, the section has been created at least in the active checkpoint replica when the invocation returns and will be created asynchronously in the other checkpoint replicas.

7.3.12 Delete a checkpoint section

Syntax:

```
SaErrorT saCkptSectionDelete(
        const SaCkptCheckpointHandleT *checkpointHandle,
        const SaCkptSectionIdT *sectionId
);
```

Parameters

checkpointHandle - [in] A pointer to the handle to the checkpoint obtained by a previous invocation of the *saCkptCheckpointOpen()* function with the SA_CKPT_CHECKPOINT_WRITE flag set.

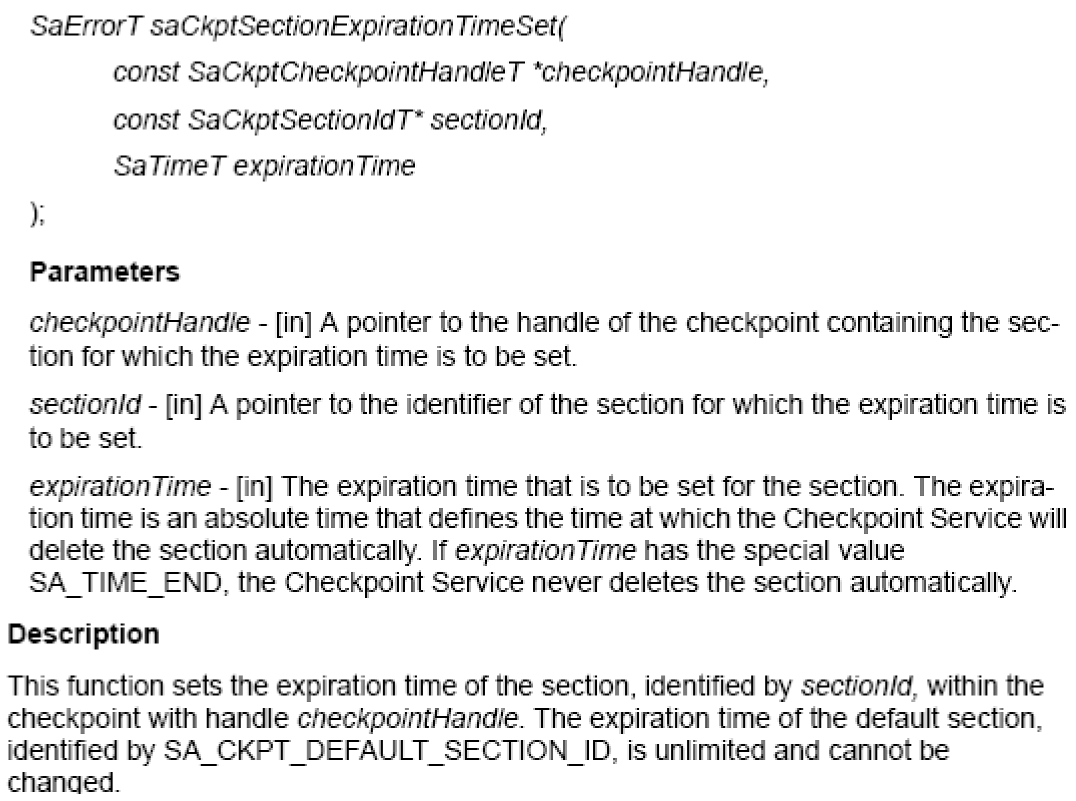
sectionId - [in] A pointer to the identifier of the section that is to be deleted.

Description

This function deletes a section. If the SA_CKPT_WR_ALL_REPLICAS property is set, the section has been deleted in all of the checkpoint replicas when the invocation returns; otherwise, the section has been deleted at least in the active checkpoint replica when the invocation returns. The default section, identified by SA_CKPT_DEFAULT_SECTION_ID, cannot be deleted by invoking the *saCkptSectionDelete()* function.

7.3.13 Set Checkpoint expiration time

Syntax:



```
SaErrorT saCkptSectionExpirationTimeSet(
        const SaCkptCheckpointHandleT *checkpointHandle,
        const SaCkptSectionIdT* sectionId,
        SaTimeT expirationTime
);
```

Parameters

checkpointHandle - [in] A pointer to the handle of the checkpoint containing the section for which the expiration time is to be set.

sectionId - [in] A pointer to the identifier of the section for which the expiration time is to be set.

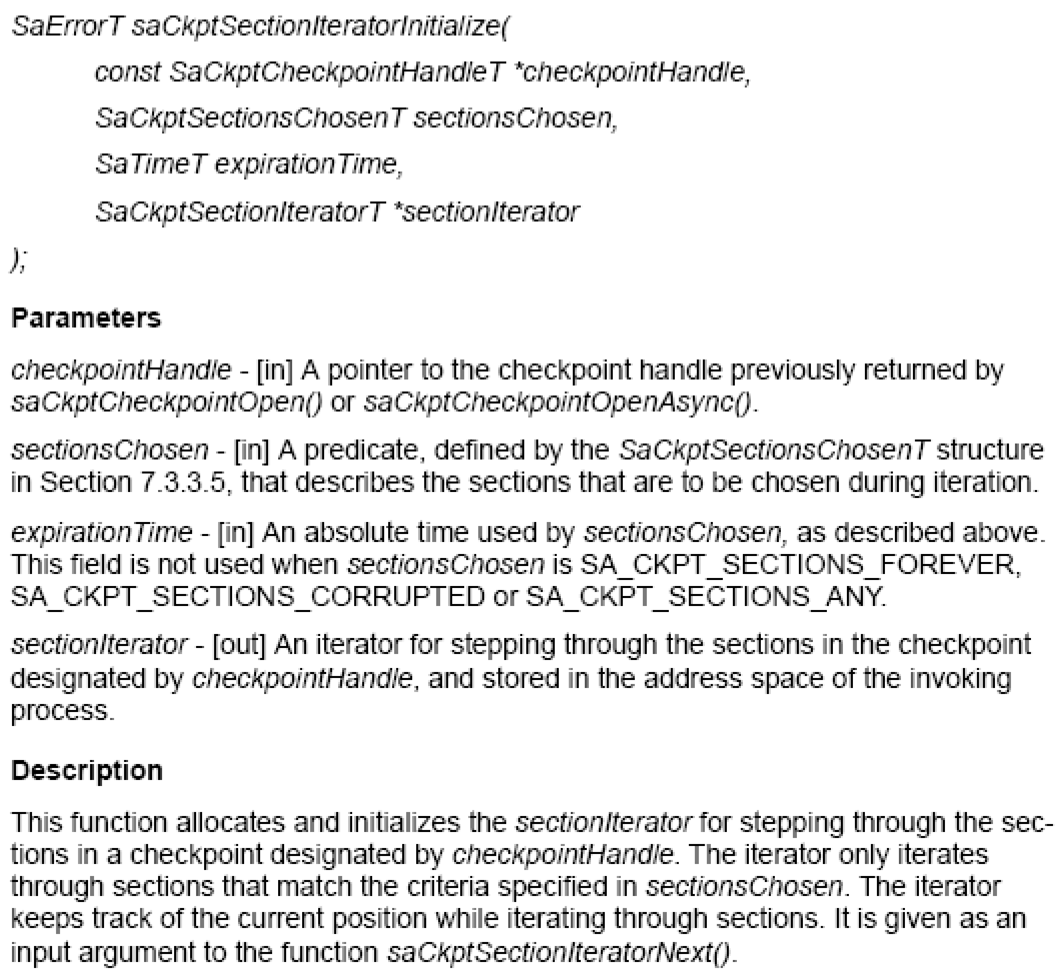
expirationTime - [in] The expiration time that is to be set for the section. The expiration time is an absolute time that defines the time at which the Checkpoint Service will delete the section automatically. If *expirationTime* has the special value SA_TIME_END, the Checkpoint Service never deletes the section automatically.

Description

This function sets the expiration time of the section, identified by *sectionId,* within the checkpoint with handle *checkpointHandle*. The expiration time of the default section, identified by SA_CKPT_DEFAULT_SECTION_ID, is unlimited and cannot be changed.

7.3.14 Iterator initialization of checkpoint

Syntax:



```
SaErrorT saCkptSectionIteratorInitialize(
        const SaCkptCheckpointHandleT *checkpointHandle,
        SaCkptSectionsChosenT sectionsChosen,
        SaTimeT expirationTime,
        SaCkptSectionIteratorT *sectionIterator
);
```

Parameters

checkpointHandle - [in] A pointer to the checkpoint handle previously returned by *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()*.

sectionsChosen - [in] A predicate, defined by the *SaCkptSectionsChosenT* structure in Section 7.3.3.5, that describes the sections that are to be chosen during iteration.

expirationTime - [in] An absolute time used by *sectionsChosen,* as described above. This field is not used when *sectionsChosen* is SA_CKPT_SECTIONS_FOREVER, SA_CKPT_SECTIONS_CORRUPTED or SA_CKPT_SECTIONS_ANY.

sectionIterator - [out] An iterator for stepping through the sections in the checkpoint designated by *checkpointHandle*, and stored in the address space of the invoking process.

Description

This function allocates and initializes the *sectionIterator* for stepping through the sections in a checkpoint designated by *checkpointHandle*. The iterator only iterates through sections that match the criteria specified in *sectionsChosen*. The iterator keeps track of the current position while iterating through sections. It is given as an input argument to the function *saCkptSectionIteratorNext()*.

7.3.15 Return next section using iteration

Syntax:

```
SaErrorT saCkptSectionIteratorNext(
        SaCkptSectionIteratorT *sectionIterator,
        SaCkptSectionDescriptorT *sectionDescriptor
);
```

Parameters

sectionIterator - [in/out] A pointer to the iterator for stepping through the sections in the checkpoint, obtained from an invocation of *saCkptSectionIteratorInitialize()*.

sectionDescriptor - [out] A pointer to a *SaCkptSectionDescriptorT* structure,

Description

This function iterates over an internal table of sections, using an iterator defined using *saCkptSectionIteratorInitialize()*. When the function returns, *sectionDescriptor* is set to the descriptor of a section, and the iterator is updated. A subsequent invocation of *saCkptSectionIteratorNext()* returns another section. When there are no more sections to return, an error is returned.

Every section created before the invocation of the *saCkptSectionIteratorInitialize()* function and not deleted before the invocation of *saCkptSectionIteratorFinalize()* shall be returned exactly once by this invocation. No other guarantees are made: sections that are created after an iterator is initialized, or deleted before an iterator is closed, may or may not be returned by an invocation of this function.

7.3.16 Clear Iterator resources attached to a checkpoint

Syntax:

```
SaErrorT saCkptSectionIteratorFinalize(
        SaCkptSectionIteratorT *sectionIterator
);
```

Parameters

sectionIterator - [in] A pointer to a structure obtained from an invocation of the *saCkptSectionIteratorInitialize()* function.

Description

This function frees resources allocated for iteration.

7.3.17 Write to checkpoint data store

Syntax:

```
- -
SaErrorT saCkptCheckpointWrite(
        const SaCkptCheckpointHandleT *checkpointHandle,
        const SaCkptIOVectorElementT *ioVector,
        SaUint32T numberOfElements,
        SaUint32T *erroneousVectorIndex
);
```

Parameters

checkpointHandle - [in] A pointer to a handle to the checkpoint that is to be written. The checkpoint handle *checkpointHandle* must be obtained by a previous invocation of the *saCkptCheckpointOpen()* function with the SA_CKPT_CHECKPOINT_WRITE flag set.

ioVector - [in] A pointer to a vector with elements *ioVector*[0], ..., *ioVector*[*numberOfElements* - 1]. Each element is of the type *SaCkptIOVectorElementT,* defined in Section 7.3.4.1, which contains the following fields: *sectionId, dataBuffer, dataSize, dataBuffer, dataOffset* and *readSize*. If *sectionId* is equal to SA_CKPT_DEFAULT_SECTION_ID, then the default section is written.*The dataSize* is at most *maxSectionSize* as specified in the creation attributes of the checkpoint. The *readSize* is not used by the *saCkptCheckpointWrite()* function.

numberOfElements - [in] Size of the *ioVector.*

erroneousVectorIndex - [out] A pointer to an index, stored in the caller's address space, of the first *iovector* element that makes the invocation fail. If the index is set to NULL or if the invocation succeeds, the field remains unchanged.

Description

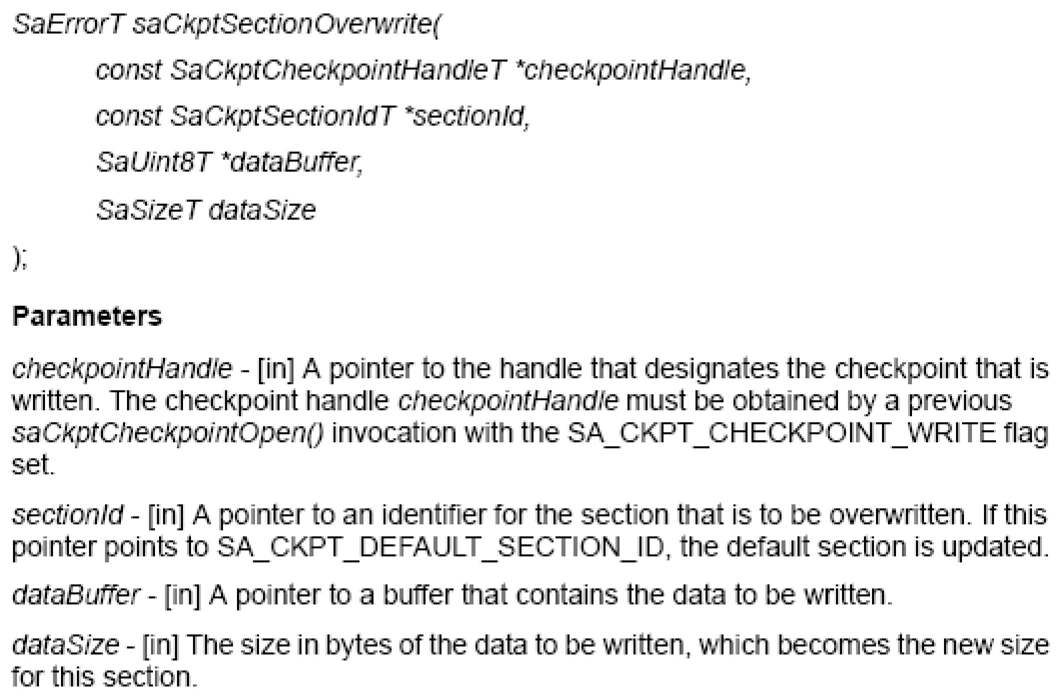
This function writes data from the memory regions specified by *ioVector* into a checkpoint:

- If this checkpoint has been created with the SA_CKPT_WR_ALL_REPLICAS property, all of the checkpoint replicas have been updated when the invocation returns. If the invocation does not complete or returns with an error, nothing has been written at all.
- If the checkpoint has been created with the SA_CKPT_WR_ACTIVE_REPLICA property, the active checkpoint replica has been updated when the invocation returns. Other checkpoint replicas are updated asynchronously. If the invocation does not complete or returns with an error, nothing has been written at all.
- If the checkpoint been created with the SA_CKPT_WR_ACTIVE_REPLICA_WEAK property, the active checkpoint replica has been updated when the invocation returns. Other checkpoint replicas are updated asynchronously. If the invocation returns with an error, nothing has been written at all. However, if the invocation does not complete, the operation may be partially completed and some sections may be corrupted in the active checkpoint replica.

If one or more HA process is trying to write to the checkpoint we expect those application to cooperatively synchronize themselves and use the checkpoint in order to ensure proper sequence of write operation.

7.3.18 Overwrite checkpoint region

Syntax:



```
SaErrorT saCkptSectionOverwrite(
        const SaCkptCheckpointHandleT *checkpointHandle,
        const SaCkptSectionIdT *sectionId,
        SaUint8T *dataBuffer,
        SaSizeT dataSize
);
```

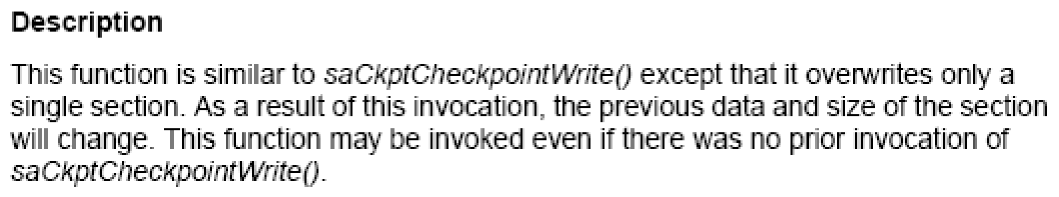
Parameters

checkpointHandle - [in] A pointer to the handle that designates the checkpoint that is written. The checkpoint handle *checkpointHandle* must be obtained by a previous *saCkptCheckpointOpen()* invocation with the SA_CKPT_CHECKPOINT_WRITE flag set.

sectionId - [in] A pointer to an identifier for the section that is to be overwritten. If this pointer points to SA_CKPT_DEFAULT_SECTION_ID, the default section is updated.

dataBuffer - [in] A pointer to a buffer that contains the data to be written.

dataSize - [in] The size in bytes of the data to be written, which becomes the new size for this section.



Description

This function is similar to *saCkptCheckpointWrite()* except that it overwrites only a single section. As a result of this invocation, the previous data and size of the section will change. This function may be invoked even if there was no prior invocation of *saCkptCheckpointWrite()*.

7.3.19 Read checkpoint store

Syntax:

SaErrorT saCkptCheckpointRead(

*const SaCkptCheckpointHandleT *checkpointHandle,*

*SaCkptIOVectorElementT *ioVector,*

SaUint32T numberOfElements,

*SaUint32T *erroneousVectorIndex*

);

Parameters

checkpointHandle - [in] A pointer to the handle to the checkpoint that is to be read.

ioVector - [in/out] A pointer to a vector that contains elements *ioVector*[0], ..., *ioVector*[*numberOfElements* - 1]. Each element is of the type *saCkptIOVectorElementT,*

- *sectionId* - [in] The identifier of the section to be written or read.
- *dataBuffer* - [in/out] A pointer to a buffer containing the data to be written or read. If *dataBuffer* is NULL, the value of *datasize* provided by the invoker is ignored and the buffer must be deallocated by the invoker.
    - *dataSize* - [in] Size of the data to be written to, or read from, the buffer *dataBuffer*. The size is at most *maxSectionSize* as specified in the creation attributes of the checkpoint.
    - *dataOffset* - [in] Offset in the section that marks the start of the data that is to be written or read.
    - *readSize* - [out] Used by *saCkptCheckpointRead()* to record the number of bytes of data that have been read; otherwise, this field is not used.

numberOfElements - [in] The size of the *ioVector*.

erroneousVectorIndex - [out] A pointer to an index, in the caller's address space, of the first vector element that causes the invocation to fail. If the invocation succeeds, then *erroneousVectorIndex* is NULL and should be ignored.

Description

This function copies data from a checkpoint replica into the vector specified by *ioVector*. Some of the buffers provided to the invocation may have been modified if the invocation does not succeed.

7.3.20 Synchronize checkpoint

Syntax:

```
SaErrorT saCkptCheckpointSynchronize(
        const SaCkptCheckpointHandleT *ckeckpointHandle,
        SaTimeT timeout
);


SaErrorT saCkptCheckpointSynchronizeAsync(
        const SaCkptHandleT *ckptHandle,
        SaInvocationT invocation,
        const SaCkptCheckpointHandleT *checkpointHandle
);
```

Parameters

ckptHandle - [in] A pointer to the handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

invocation - [in] This parameter designates a particular invocation of the response callback.

checkpointHandle -[in] A pointer to the handle of the checkpoint that is to be synchronized.

timeout - [in] The synchronous version shall terminate if the time it takes exceeds *timeout*. However, the propagation of the checkpoint data to other checkpoint replicas might continue even if this error is returned.

Description

The *saCkptCheckpointSynchronize()* and *saCkptCheckpointSynchronizeAsync()* function ensures that all previous operations applied on the active checkpoint replica are propagated to other checkpoint replicas. Such operations are *saCkptCheckpointWrite()*, *saCkptSectionOverwrite()*, *saCkptSectionCreate()* and *saCkptSectionDelete()*.

These invocations are useful only for checkpoints created with the synchronization values SA_CKPT_WR_ACTIVE_REPLICA or SA_CKPT_WR_ACTIVE_REPLICA_WEAK.

Only those processes that have the checkpoint open in SA_CKPT_CHECKPOINT_WRITE mode may invoke this function.

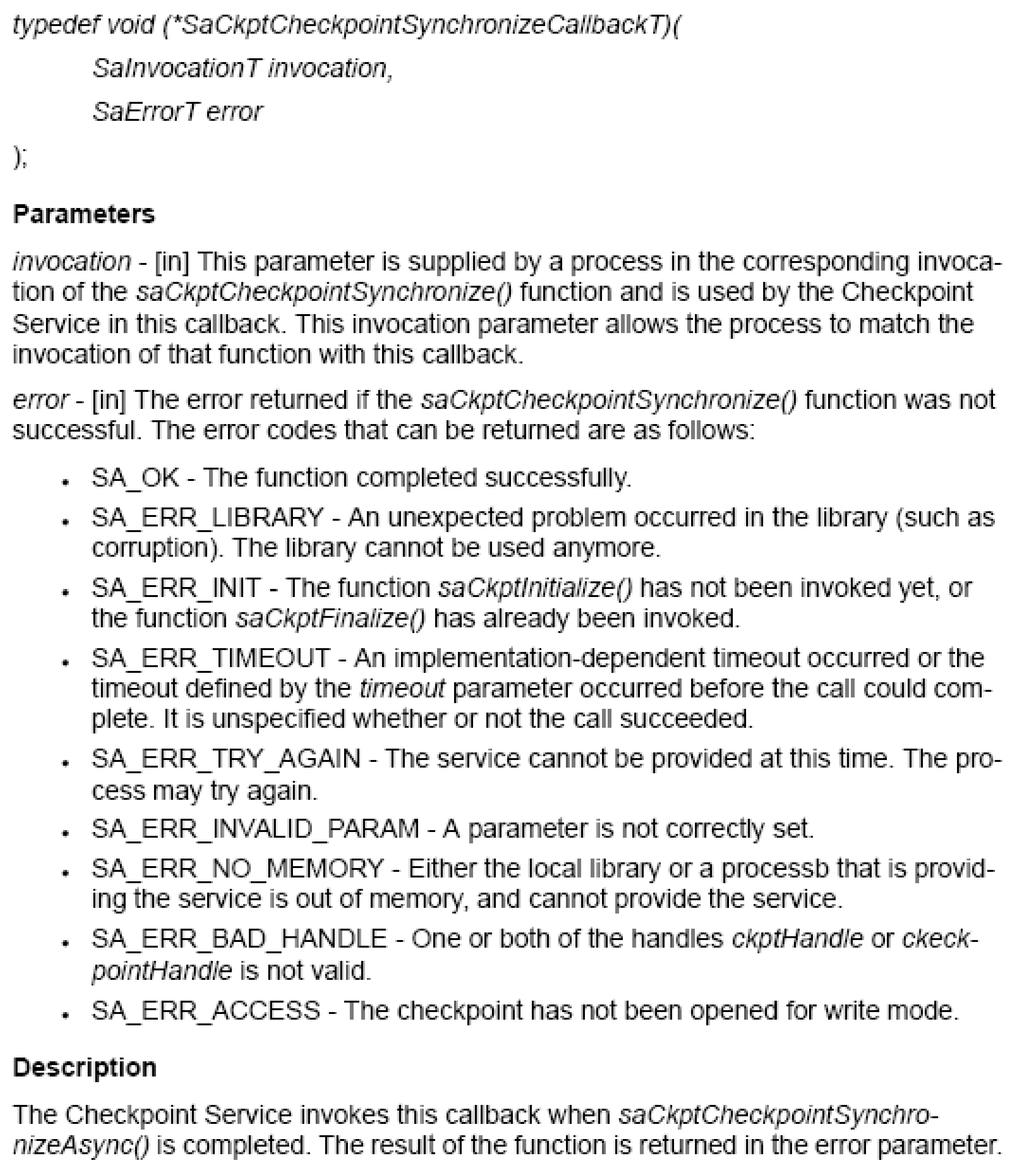
After successful completion of this function, all checkpoint replicas should be identical.

For the *saCkptCheckpointSynchronize()* function, when the timeout expires, there is no guarantee whether or not the checkpoint replicas have been synchronized.

For the *saCkptCheckpointSynchronizeAsync()* function, completion of the function is signaled by the associated *saCkptCheckpointSynchronizeCallback()* function. The invoking process sets the *invocation* parameter and the Checkpoint Service uses the value of *invocation* in the invocation of the callback function.

7.3.21 Checkpoint synchronize call back function

Syntax:



```
typedef void (*SaCkptCheckpointSynchronizeCallbackT)(
        SaInvocationT invocation,
        SaErrorT error
);
```

Parameters

invocation - [in] This parameter is supplied by a process in the corresponding invocation of the *saCkptCheckpointSynchronize()* function and is used by the Checkpoint Service in this callback. This invocation parameter allows the process to match the invocation of that function with this callback.

error - [in] The error returned if the *saCkptCheckpointSynchronize()* function was not successful. The error codes that can be returned are as follows:

- SA_OK - The function completed successfully.
- SA_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.
- SA_ERR_INIT - The function *saCkptInitialize()* has not been invoked yet, or the function *saCkptFinalize()* has already been invoked.
- SA_ERR_TIMEOUT - An implementation-dependent timeout occurred or the timeout defined by the *timeout* parameter occurred before the call could complete. It is unspecified whether or not the call succeeded.
- SA_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may try again.
- SA_ERR_INVALID_PARAM - A parameter is not correctly set.
- SA_ERR_NO_MEMORY - Either the local library or a processb that is providing the service is out of memory, and cannot provide the service.
- SA_ERR_BAD_HANDLE - One or both of the handles *ckptHandle* or *ckeckpointHandle* is not valid.
- SA_ERR_ACCESS - The checkpoint has not been opened for write mode.

Description

The Checkpoint Service invokes this callback when *saCkptCheckpointSynchronizeAsync()* is completed. The result of the function is returned in the error parameter.

7.4 Event Service API

If HA applications or HA-SAPI or HA-FAPI needs to communicate across cards (that is between standby and active) they can use HA Event Service API. The main purpose is to exchange HA application specific state transition or other HA application specific events to group of HA resources that are belonging to same service group or managed under same HA middleware.

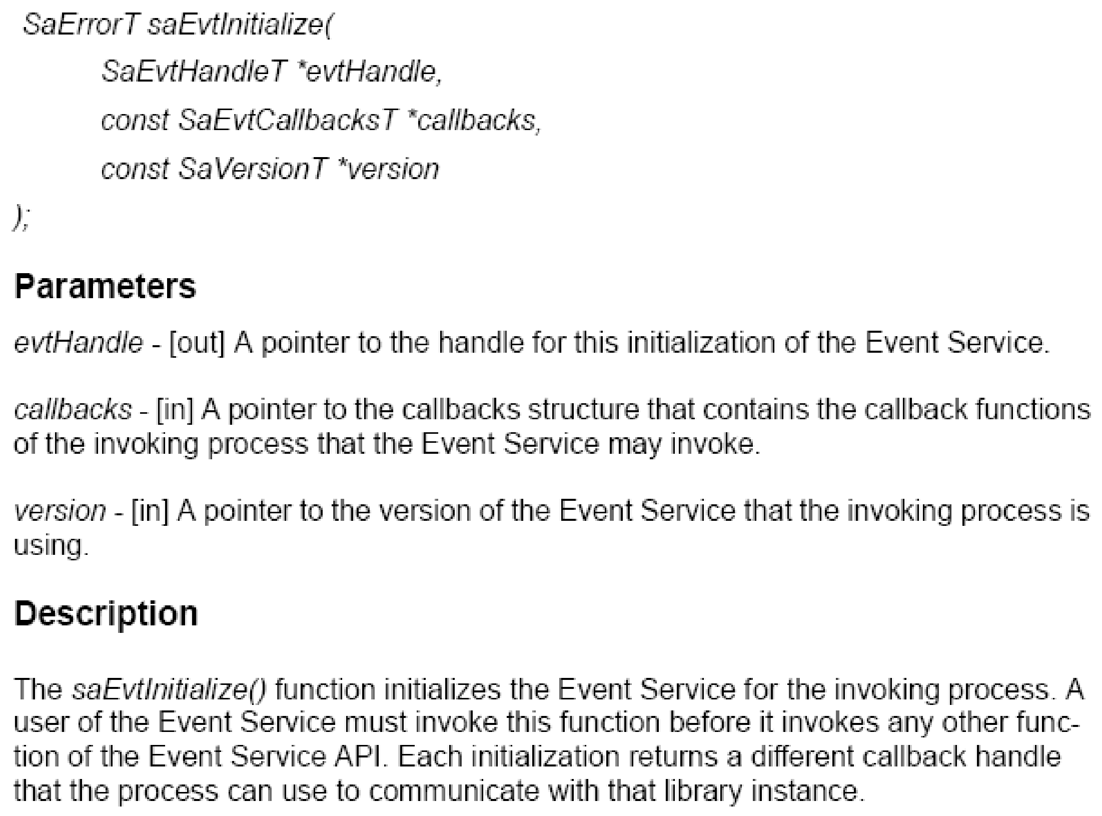
7.4.1 Usage Model

This is a optional service and depending upon the nature of application either HA aware application or HA-SAPI or HA-FAPI may use this. Each HA implementation must implement this.

- HA application that wishes to send or receive events needs to subscribe the HA event service.
- Multiple subscribers can subscribe to same event and HA event service implementation will post appropriate event when some one publish an events. HA implementation supports several logical channels for posting events and each channel can be categorized as
  - best effort delivery
  - at most delivery service
  - Event priority
  - Event completeness
  - Retention time and persistence
- HA Event service depends on AMF implementation in NPF environment.

7.4.2 Initialize Event Service

Syntax:



```
SaErrorT saEvtInitialize(
        SaEvtHandleT *evtHandle,
        const SaEvtCallbacksT *callbacks,
        const SaVersionT *version
);
```

Parameters

evtHandle - [out] A pointer to the handle for this initialization of the Event Service.

callbacks - [in] A pointer to the callbacks structure that contains the callback functions of the invoking process that the Event Service may invoke.

version - [in] A pointer to the version of the Event Service that the invoking process is using.

Description

The *saEvtInitialize()* function initializes the Event Service for the invoking process. A user of the Event Service must invoke this function before it invokes any other function of the Event Service API. Each initialization returns a different callback handle that the process can use to communicate with that library instance.

Each HA aware resource needs to initialize the Event service library in order to make use of the event service. If HA-SAPI or HA-FAPI wants to subscribe or publish events, then it will invoke this call and will install appropriate callback functions. If both HA aware resource and HA-SAPI or HA-FAPI is required to use the event service, then HA aware resource should initialize the library. HA implementation must return a unique handle each time when this checkpoint service is being initialized.

7.4.3 Installing event dispatch mechanism

Syntax:

```
 SaErrorT saEvtDispatch(
        const SaEvtHandleT *evtHandle,
        SaDispatchFlagsT dispatchFlags
);
```

Parameters

evtHandle - [in] A pointer to the handle, obtained through the *saEvtInitialize()* function, designating this particular initialization of the Event Service.

dispatchFlags - [in] Flags that specify the callback execution behavior of the the *saEvtDispatch()* function, which have the values SA_DISPATCH_ONE, SA_DISPATCH_ALL or SA_DISPATCH_BLOCKING, as defined in Section 3.3.8.

Description

The *saEvtDispatch()* function invokes, in the context of the calling thread, one or all of the pending callbacks for the handle *evtHandle*.

7.4.4 Open an Event Channel

Syntax:

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

Parameters

evtHandle - [in] A pointer to the handle, obtained through the *saEvtInitialize()* function, designating this particular initialization of the Event Service.

channelName - [in] A pointer to the name of the event channel, which globally identifies an event channel in a cluster. If the name is already in use within the cluster, the error SA_ERR_EXIST will be returned.

channelOpenFlags - [in] The requested access modes of the event channel. The value of this parameter is obtained by a bitwise OR of the SA_EVT_CHANNEL_PUBLISHER, SA_EVT_CHANNEL_SUBSCRIBER and SA_EVT_CHANNEL_CREATE flags defined by *SaEvtChannelOpenFlagsT* in Section 8.3.3. If SA_EVT_CHANNEL_PUBLISHER is set, the process may use the returned event channel handle with *saEvtEventPublish()*. If SA_EVT_CHANNEL_SUBSCRIBER is set, the process may use the returned event channel handle with *saEvtEventSubscribe()*. If SA_EVT_CHANNEL_CREATE is set, the Event Service creates an event channel if one does not already exist.

channelHandle - [in/out] A pointer to the handle of the event channel, provided by the invoking process in the address space of the process. If the event channel is opened successfully, the Event Service stores, in *channelHandle,* the handle that the process uses to access the channel in subsequent invocations of the functions of the Event Service API.

Description

The *saEvtChannelOpen()* function creates a new event channel or open an existing channel. The *saEvtChannelOpen()* function is a blocking operation and returns a new event channel handle.

An event channel may be opened multiple times by the same or different processes for publishing, and subscribing to, events. If a process opens an event channel multiple times, it is possible to receive the same event multiple times. However, a process shall never receive an event more than once on a particular event channel handle.

If a process opens a channel twice and an event is matched on both open channels, the Event Service performs two callbacks -- one for each opened channel.

7.4.5 Close event channel

Syntax:

```
SaErrorT saEvtChannelClose(
        SaEvtChannelHandleT *channelHandle
);
```

Parameters

channelHandle - [in] A pointer to the handle of the event channel to close.

Description

The *saEvtChannelClose()* function closes an event channel and frees resources allocated for that event channel in the invoking process. If the event channel is not referenced by any process and does not hold any events with non-zero retention time, the Event Service automatically deletes the event channel

7.4.6 Set attribute for Events

Syntax:

```
SaErrorT saEvtEventAttributesSet(
        const SaEvtEventHandleT *eventHandle,
        const SaEvtEventPatternArrayT *patternArray,
        SaUint8T priority,
        SaTimeT retentionTime,
        const SaNameT *publisherName
);
```

Parameters

eventHandle - [in] A pointer to the handle of the event whose attributes are to be set.

patternArray - [in] A pointer to a structure that contains the array of patterns to be placed into the event pattern array and the number of such patterns.

priority - [in] The priority of the event.

retentionTime - [in] The duration for which the event will be retained.

publisherName - [in] A pointer to the name of the publisher of the event.

Description

This function may be used to assign writeable event attributes. It takes as arguments an event handle *eventHandle* and the attribute to be set in the event structure of the event with that handle. Note: The only attributes that a process can set are the *patter-nArray*, *priority*, *retentionTime* and *publisherName* attributes.

7.4.7 Get Event attributes

Syntax:

```
SaErrorT saEvtEventAttributesGet(
        const SaEvtChannelHandleT *channelHandle,
        const SaEvtEventHandleT *eventHandle,
        SaEvtEventPatternArrayT *patternArray,
        SaUint8T *priority,
        SaTimeT *retentionTime,
        SaNameT *publisherName,
        SaClmNodeIdT *publisherNodeId,
        SaTimeT *publishTime,
        SaEvtEventIdT *eventId
);
```

Parameters

channelHandle - [in] A pointer to the handle of the event channel on which the event data is to be received. This parameter is an event channel handle returned from a previous call to the *saEvtChannelOpen()* function.

eventHandle - [in] A pointer to the handle of the event whose attributes are to be retrieved.

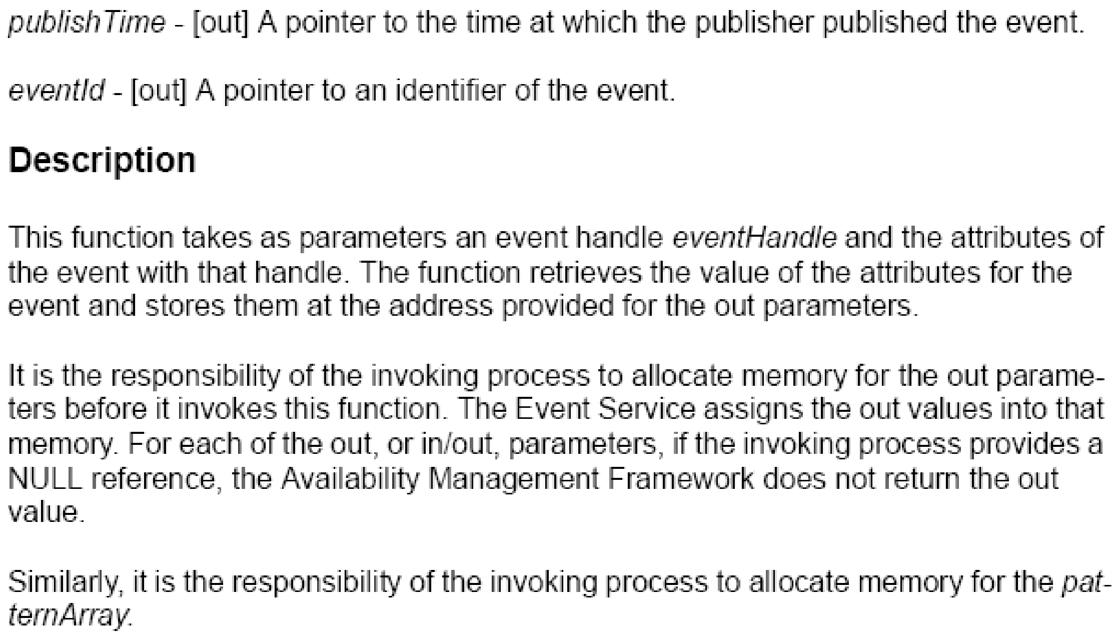
patternArray - [in/out] A pointer to a structure that contains the array of patterns to be retrieved from the event pattern array and the number of such patterns. A process that invokes this function provides the *patternArray*, and the Event Service inserts the patterns into the successive entries of the *patternArray*, starting with the first entry and continuing until the patterns are exhausted. The number of patterns that the Event Service inserts into the *patternArray* is the minimum of the number of patterns for the event and the *patternsNumber* value of the in value of *patternArray,* supplied by the process. If there are more patterns than *patternsNumber*, the Event Service does not insert those additional patterns. The *pattternsNumbe*r value of the out value of *patternArray*, supplied by the Event Service, can be less than, equal to, or greater than the value of *patternsNumber,* supplied by the process.

priority - [out] A pointer to the priority of the event.

retentionTime - [out] A pointer to the duration for which the publisher will retain the event.

publisherName - [out] A pointer to the name of the publisher of the event.

publisherNodeId- [out] A pointer to the identifier of the card from which the events was published.



publishTime - [out] A pointer to the time at which the publisher published the event.

eventId - [out] A pointer to an identifier of the event.

Description

This function takes as parameters an event handle *eventHandle* and the attributes of the event with that handle. The function retrieves the value of the attributes for the event and stores them at the address provided for the out parameters.

It is the responsibility of the invoking process to allocate memory for the out parameters before it invokes this function. The Event Service assigns the out values into that memory. For each of the out, or in/out, parameters, if the invoking process provides a NULL reference, the Availability Management Framework does not return the out value.

Similarly, it is the responsibility of the invoking process to allocate memory for the *patternArray*.

7.4.8 Get Event Data

Syntax:

```
SaErrorT saEvtEventDataGet(
        const SaEvtEventHandleT *eventHandle,
        void *eventData,
        SaSizeT *eventDataSize
);
```

Parameters

eventHandle - [in] A pointer to the handle to the event previously delivered by *saE-vtEventDeliverCallback()*.

eventData - [in/out] A pointer to a buffer provided by the process in which the Event Service stores the data associated with the delivered event.

eventDataSize - [in/out] The in value of *eventDataSize* is the size of the *eventData* buffer provided by the invoking process. The Event Service must not write more than *eventDataSize* bytes into the *eventData* buffer. The out value of *eventDataSize* is the size of the *eventData* of the event, which may be less than, equal to, or greater than the in value of *eventDataSize*. Note: An eventData buffer of size SA_EVT_DATA_MAX_LEN bytes or more will always be able to contain the largest possible event data associated with an event.

Description

The *saEvtEventDataGet()* function allows a process to retrieve the data associated with an event previously delivered by *saEvtEventDeliverCallback()*.

7.4.9 Event Delivery call back

Syntax:

```
typedef void(*SaEvtEventDeliverCallbackT)(
        const SaEvtChannelHandleT *channelHandle,
        SaEvtSubscriptionIdT subscriptionId,
        const SaEvtEventHandleT *eventHandle,
        const SaSizeT eventDataSize
);
```

Parameters

channelHandle - [in] A pointer to the handle to the event channel on which the event has been received.

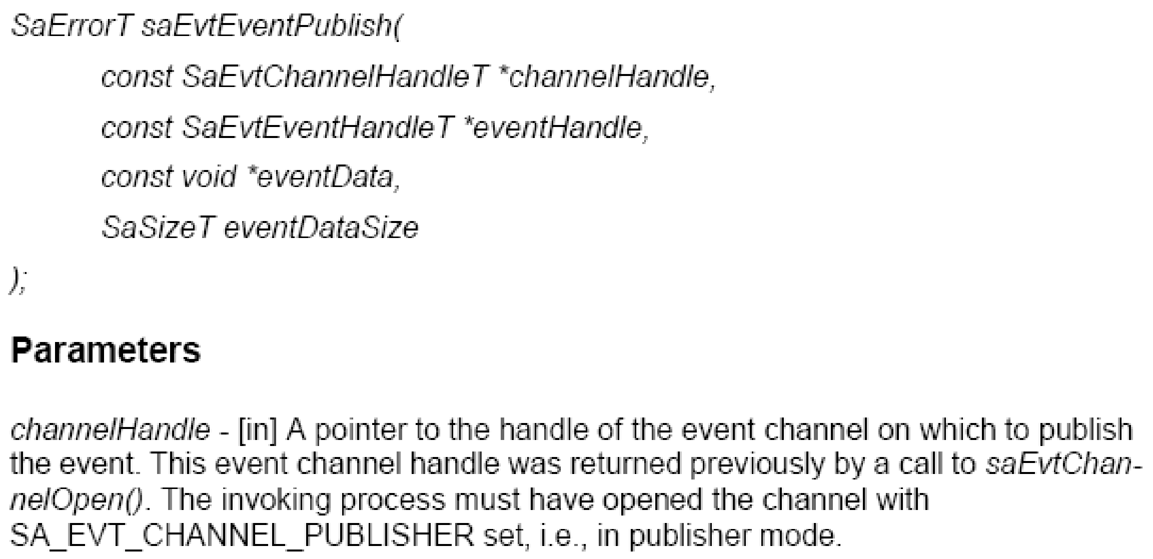
subscriptionId - [in] An identifier that a process supplied in an *saEvtEventSubscribe()* invocation that enables it to determine which subscription resulted in the delivery of the event.

eventHandle - [in] A pointer to the handle to the event that is allocated by the Event Service before it invokes this callback. The Event Service must not deallocate the memory space for the handle before the process invokes the *saEvtEventFree()* function.

eventDataSize - [in] The size of the data associated with the event.

7.4.10 Publish an event

Syntax:



```
SaErrorT saEvtEventPublish(
        const SaEvtChannelHandleT *channelHandle,
        const SaEvtEventHandleT *eventHandle,
        const void *eventData,
        SaSizeT eventDataSize
);
```

Parameters

channelHandle - [in] A pointer to the handle of the event channel on which to publish the event. This event channel handle was returned previously by a call to *saEvtChannelOpen()*. The invoking process must have opened the channel with SA_EVT_CHANNEL_PUBLISHER set, i.e., in publisher mode.

eventHandle - [in] A pointer to the handle of the event that is to be published. The event must have been allocated by *saEvtEventAllocate()* and the patterns must have been set by *saEvtEvenPatternArraySet()*.

eventData - [in] A pointer to a buffer that contains additional event information for the event being published. This parameter is set to NULL if no additional information is associated with the event. The process may deallocate the memory for *eventData* when *saEvtEventPublish()* returns.

eventDataSize - [in] The number of bytes in the buffer pointed to by *eventData*. Setting *eventDataSize* greater than SA_EVENT_DATA_MAX_SIZE results in only the first SA_EVENT_DATA_MAX_SIZE characters being published.

Description

The *saEvtEventPublish()* function publishes an event on the channel designated by *channelHandle*. The event to be published consists of a standard set of attributes (the event header) and an optional data part.

Before an event can be published, the publisher process must invoke the *saEvtEventPatternArraySet()* function to set the event patterns. The event is delivered to subscribers whose subscription filter matches the event patterns.

When the Event Service publishes an event, it automatically sets the following read-only event attributes:

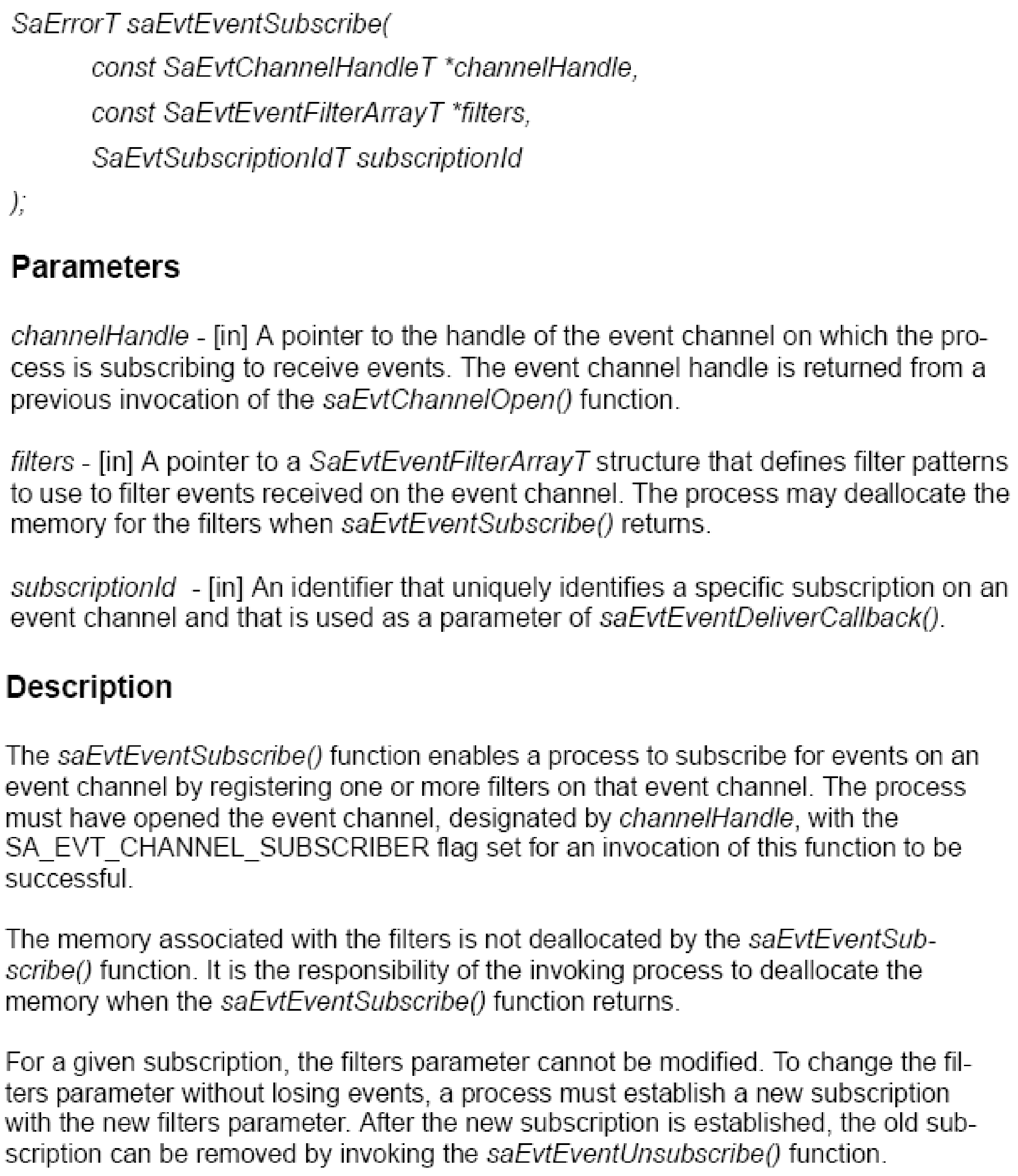
- Event attribute time
- Event publisher identifier
- Event publisher node identifier
- Event identifier

In addition to the event attributes, a process can supply values for the *eventData* and *eventDataSize* parameters for publication as part of the event. The data portion of the event may be at most SA_EVT_DATA_MAX_LEN bytes in length.

The process may assume that the invocation of *saEvtEventPublish()* copies all pertinent parameters, including the memory associated with the *eventHandle* and *eventData* parameters, to its own local memory. However, the invocation does not automatically deallocate memory associated with the *eventHandle* and *eventData* parameters. It is the responsibility of the invoking process to deallocate the memory for those parameters after *saEvtEventPublish()* returns.

7.4.11 Subscribe to an Event

Syntax:



```
SaErrorT saEvtEventSubscribe(
        const SaEvtChannelHandleT *channelHandle,
        const SaEvtEventFilterArrayT *filters,
        SaEvtSubscriptionIdT subscriptionId
);
```

Parameters

channelHandle - [in] A pointer to the handle of the event channel on which the process is subscribing to receive events. The event channel handle is returned from a previous invocation of the *saEvtChannelOpen()* function.

filters - [in] A pointer to a *SaEvtEventFilterArrayT* structure that defines filter patterns to use to filter events received on the event channel. The process may deallocate the memory for the filters when *saEvtEventSubscribe()* returns.

subscriptionId - [in] An identifier that uniquely identifies a specific subscription on an event channel and that is used as a parameter of *saEvtEventDeliverCallback()*.

Description

The *saEvtEventSubscribe()* function enables a process to subscribe for events on an event channel by registering one or more filters on that event channel. The process must have opened the event channel, designated by *channelHandle*, with the SA_EVT_CHANNEL_SUBSCRIBER flag set for an invocation of this function to be successful.

The memory associated with the filters is not deallocated by the *saEvtEventSubscribe()* function. It is the responsibility of the invoking process to deallocate the memory when the *saEvtEventSubscribe()* function returns.

For a given subscription, the filters parameter cannot be modified. To change the filters parameter without losing events, a process must establish a new subscription with the new filters parameter. After the new subscription is established, the old subscription can be removed by invoking the *saEvtEventUnsubscribe()* function.

7.4.12 Unsubscribe to an event

Syntax:

```
SaErrorT saEvtEventUnsubscribe(
        const SaEvtChannelHandleT *channelHandle,
        SaEvtSubscriptionIdT subscriptionId
);
```

Parameters

channelHandle - [in] A pointer to the event channel for which the subscriber is requesting the Event Service to delete the subscription.

subscriptionId - [in] The identifier of the subscription that the subscriber is requesting the Event Service to delete.

Description

The *saEvtEventUnsubscribe()* function allows a process to stop receiving events for a particular subscription on an event channel by removing the subscription. The *saE-vtEventUnsubscribe()* operation is successful if the *subscriptionId* parameter matches a previously registered subscription. Pending events that no longer match any sub-scription, because the *saEvtEventUnsubscribe()* operation had been invoked, are purged. a process that wishes to modify a subscription without losing any events must establish the new subscription before removing the existing subscription.

8 Appendix

8.1 Header file definition

8.1.1 NPF Extension API

```
/*
 * This header file defines typedefs, constants and
 * functions of the NPF High availability extension API
 *
 * This assumes that the definitions common to all NPF APIs
 * are available in a separate manner (a different header
 * file, etc.)
 */

#ifndef __NPF_HA_API_H_
#define __NPF_HA_API_H_

#ifdef __cplusplus
extern "C" {
#endif

typedef NPF_char8_t    NPF_HA_Component_Name_t;  /*Holds the name of the
HA component */

typedef NPF_char8_t    NPF_HA_Role; /*Defines the HA role and mode*/

typedef  NPF_int32_t    NPF_HA_Correlator_t;     /*Correlates     application
instance */

typedef  NPF_int32_t    NPF_HA_Instance_Id; /*identifies HA instance */


/****************************************************
 *      HA Extension API FUNCTION CALLS            *
 ****************************************************/
NPF_error_t  NPF_HAInit (
   NPF_IN   NPF_HA_Component_Name *component_Name;
   NPF_IN   NPF_HA_Role     role,
   NPF_IN   NPF_HA_Correlator_t correlator,
   NPF_IN   NPF_HA_Instance_Id  instance_Id,
   NPF_IN   NPF_boolean_t   mode,
);

void   NPF_HADeregister(
   NPF_IN   NPF_HA_Component_Name *component_Name;
   NPF_IN   NPF_HA_Correlator_t correlator,
   NPF_IN   NPF_HA_Instance_Id  instance_Id,
);

#ifdef __cplusplus
}
#endif
```

```
#endif /* __NPF_HA_API_H_ */
```

8.1.2 SAForum API

```
/*
This header file is based on AIS document SAI-AIS-A.01.01
This include prototypes that are required and needs to be
supported in NPF-SW-HA environment.

/*
  In order to compile, all opaque types that appear as <...> in
  the spec have been defined as OPAQUE_TYPE (which is an
integer).


typedef OPAQUE_TYPE SaInvocationT;
typedef OPAQUE_TYPE SaSizeT;
typedef OPAQUE_TYPE SaOffsetT;
typedef OPAQUE_TYPE SaSelectionObjectT;
typedef OPAQUE_TYPE SaAmfHandleT;
typedef OPAQUE_TYPE SaCkptHandleT;
typedef OPAQUE_TYPE SaCkptCheckpointHandleT;
typedef OPAQUE_TYPE SaCkptSectionIteratorT;
typedef OPAQUE_TYPE SaEvtHandleT;
typedef OPAQUE_TYPE SaEvtEventHandleT;
typedef OPAQUE_TYPE SaEvtChannelHandleT;

*/


#define OPAQUE_TYPE  int

typedef OPAQUE_TYPE SaInvocationT;
typedef OPAQUE_TYPE SaSizeT;
typedef OPAQUE_TYPE SaOffsetT;
typedef OPAQUE_TYPE SaSelectionObjectT;

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

typedef char                     SaInt8T;
typedef short                    SaInt16T;
typedef long                     SaInt32T;
typedef long long                SaInt64T;
typedef unsigned char            SaUint8T;
typedef unsigned short           SaUint16T;
typedef unsigned long            SaUint32T;
typedef unsigned long long       SaUint64T;
typedef SaInt64T                 SaTimeT;

#define SA_MAX_NAME_LENGTH 256

typedef struct {
    SaUint16T length;
    unsigned char value[SA_MAX_NAME_LENGTH];
} SaNameT;

/*

NPF specific: We need to have our own versioning in order to
distinguish from normal SAForum  complaint implementation
```

```
*/
typedef struct {
    char     releaseCode;
    unsigned char major;
    unsigned char minor;
} SaVersionT;

#define SA_TRACK_CURRENT 0x01
#define SA_TRACK_CHANGES 0x02
#define SA_TRACK_CHANGES_ONLY 0x04

typedef enum {
    SA_DISPATCH_ONE = 1,
    SA_DISPATCH_ALL = 2,
    SA_DISPATCH_BLOCKING = 3
} SaDispatchFlagsT;

typedef enum {
    SA_OK = 1,
    SA_ERR_LIBRARY = 2,
    SA_ERR_VERSION = 3,
    SA_ERR_INIT = 4,
    SA_ERR_TIMEOUT = 5,
    SA_ERR_TRY_AGAIN = 6,
    SA_ERR_INVALID_PARAM = 7,
    SA_ERR_NO_MEMORY = 8,
    SA_ERR_BAD_HANDLE = 9,
    SA_ERR_BUSY = 10,
    SA_ERR_ACCESS = 11,
    SA_ERR_NOT_EXIST = 12,
    SA_ERR_NAME_TOO_LONG = 13,
    SA_ERR_EXIST = 14,
    SA_ERR_NO_SPACE = 15,
    SA_ERR_INTERRUPT =16,  /* Not supported */
    SA_ERR_SYSTEM = 17,
    SA_ERR_NAME_NOT_FOUND = 18,
    SA_ERR_NO_RESOURCES = 19,
    SA_ERR_NOT_SUPPORTED = 20,
    SA_ERR_BAD_OPERATION = 21,
    SA_ERR_FAILED_OPERATION = 22,
    SA_ERR_MESSAGE_ERROR = 23,
    SA_ERR_NO_MESSAGE = 24,
    SA_ERR_QUEUE_FULL = 25,
    SA_ERR_QUEUE_NOT_AVAILABLE = 26,
    SA_ERR_BAD_CHECKPOINT = 27,
    SA_ERR_BAD_FLAGS = 28
} SaErrorT;

/*
 * AMF related data types
*/

typedef OPAQUE_TYPE SaAmfHandleT;

typedef enum {
    SA_AMF_HEARTBEAT = 1,  /* Recommended value in NPF */
    SA_AMF_HEALTHCHECK_LEVEL1 = 2, /*Not used in NPF */
    SA_AMF_HEALTHCHECK_LEVEL2 = 3, /* Not used in NPF */
    SA_AMF_HEALTHCHECK_LEVEL3 = 4 /* Not used in NPF */
} SaAmfHealthcheckT;

typedef enum {
    SA_AMF_OUT_OF_SERVICE = 1,
    SA_AMF_IN_SERVICE = 2,
    SA_AMF_STOPPING = 3
} SaAmfReadinessStateT;
```

```
typedef enum {
    SA_AMF_ACTIVE = 1,
    SA_AMF_STANDBY = 2,
    SA_AMF_QUIESCED = 3   /* We need to add one more type
STOPPED in NPF */
} SaAmfHAStateT;


/* NPF: We need not support all redundancy model, its
application and implemenation dependent */

typedef enum {
    SA_AMF_COMPONENT_CAPABILITY_X_ACTIVE_AND_Y_STANDBY= 1,
    SA_AMF_COMPONENT_CAPABILITY_X_ACTIVE_OR_X_STANDBY = 2,
    SA_AMF_COMPONENT_CAPABILITY_1_ACTIVE_OR_Y_STANDBY = 3,
    SA_AMF_COMPONENT_CAPABILITY_1_ACTIVE_OR_1_STANDBY = 4,
    SA_AMF_COMPONENT_CAPABILITY_X_ACTIVE = 5,
    SA_AMF_COMPONENT_CAPABILITY_1_ACTIVE = 6,
    SA_AMF_COMPONENT_CAPABILITY_NO_STATE = 7
} SaAmfComponentCapabilityModelT;


/*

In NPF each resource is referred in HA environment under a
component name. Component Service Instance and Component Name
refers are same in NPF.

*/
#define SA_AMF_CSI_ADD_NEW_INSTANCE 0X1
#define SA_AMF_CSI_ALL_INSTANCES 0X2

typedef SaUint32T SaAmfCSIFlagsT;


#define SA_AMF_SWITCHOVER_OPERATION 0X1
#define SA_AMF_SHUTDOWN_OPERATION 0X2
typedef SaUint32T SaAmfPendingOperationFlagsT;

typedef struct {
    SaNameT compName;
    SaAmfReadinessStateT readinessState;
    SaAmfHAStateT haState;
} SaAmfProtectionGroupMemberT;

typedef enum {
    SA_AMF_PROTECTION_GROUP_NO_CHANGE = 1,
    SA_AMF_PROTECTION_GROUP_ADDED = 2,
    SA_AMF_PROTECTION_GROUP_REMOVED = 3,
    SA_AMF_PROTECTION_GROUP_STATE_CHANGE = 4
} SaAmfProtectionGroupChangesT;

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

typedef enum {
    SA_AMF_COMMUNICATION_ALARM_TYPE = 1,
    SA_AMF_QUALITY_OF_SERVICE_ALARM_TYPE = 2,
    SA_AMF_PROCESSING_ERROR_ALARM_TYPE = 3,
    SA_AMF_EQUIPMENT_ALARM_TYPE = 4,
    SA_AMF_ENVIRONMENTAL_ALARM_TYPE = 5
} SaAmfErrorReportTypeT;

typedef enum {
    SA_AMF_APPLICATION_SUBSYSTEM_FAILURE = 1,
    SA_AMF_BANDWIDTH_REDUCED = 2,
    SA_AMF_CALL_ESTABLISHMENT_ERROR = 3,
    SA_AMF_COMMUNICATION_PROTOCOL_ERROR = 4,
    SA_AMF_COMMUNICATION_SUBSYSTEM_FAILURE = 5,
    SA_AMF_CONFIGURATION_ERROR = 6,
    SA_AMF_CONGESTION = 7,        /* Not used in NPF */
    SA_AMF_CORRUPT_DATA = 8,
    SA_AMF_CPU_CYCLES_LIMIT_EXCEEDED = 9,/* Not used in NPF */
    SA_AMF_EQUIPMENT_MALFUNCTION = 10,
    SA_AMF_FILE_ERROR = 11,
    SA_AMF_IO_DEVICE_ERROR = 12,
    SA_AMF_LAN_ERROR, SA_AMF_OUT_OF_MEMORY = 13,
    SA_AMF_PERFORMANCE_DEGRADED = 14,
    SA_AMF_PROCESSOR_PROBLEM = 15,   /* Not used in NPF */
    SA_AMF_RECEIVE_FAILURE = 16,
    SA_AMF_REMOTE_NODE_TRANSMISSION_ERROR = 17,
    SA_AMF_RESOURCE_AT_OR_NEARING_CAPACITY = 18,
    SA_AMF_RESPONSE_TIME_EXCESSIVE = 19,
    SA_AMF_RETRANSMISSION_RATE_EXCESSIVE = 20,
    SA_AMF_SOFTWARE_ERROR = 21,
    SA_AMF_SOFTWARE_PROGRAM_ABNORMALLY_TERMINATED = 22,
    SA_AMF_SOFTWARE_PROGRAM_ERROR = 23,
    SA_AMF_STORAGE_CAPACITY_PROBLEM = 24,
    SA_AMF_TIMING_PROBLEM = 25,
    SA_AMF_UNDERLYING_RESOURCE_UNAVAILABLE = 26,
    SA_AMF_INTERNAL_ERROR = 27,
    SA_AMF_NO_SERVICE_ERROR = 28,
    SA_AMF_SOFTWARE_LIBRARY_ERROR = 29
} SaAmfProbableCauseT;

typedef enum {
    SA_AMF_CLEARED = 1,
    SA_AMF_NO_IMPACT = 2,
    SA_AMF_INDETERMINATE = 3,
    SA_AMF_CRITICAL = 4,
    SA_AMF_MAJOR = 5,
    SA_AMF_WEDGED_COMPONENT_FAILURE = 6,
    SA_AMF_COMPONENT_TERMINATED_FAILURE= 7,
    SA_AMF_NODE_FAILURE = 8,
    SA_AMF_MINOR = 9,
    SA_AMF_WARNING = 10
} SaAmfErrorImpactAndSeverityT;

typedef enum {
    SA_AMF_NO_RECOMMENDATION = 1,
    SA_AMF_INTERNALLY_RECOVERED = 2,
    SA_AMF_COMPONENT_RESTART = 3,
    SA_AMF_COMPONENT_FAILOVER = 4,
    SA_AMF_NODE_SWITCHOVER = 5,
    SA_AMF_NODE_FAILOVER = 6,
    SA_AMF_NODE_FAILFAST = 7,
    SA_AMF_CLUSTER_RESET = 8
} SaAmfRecommendedRecoveryT;

#define SA_AMF_OPAQUE_BUFFER_SIZE_MAX 256

typedef struct {
    char *buffer;
    SaSizeT size;
} SaAmfErrorBufferT;

typedef struct {
    SaAmfErrorBufferT *specificProblem;
    SaAmfErrorBufferT *additionalText;
    SaAmfErrorBufferT *additionalInformation;
} SaAmfAdditionalDataT;
```

```
typedef struct {
    SaAmfErrorReportTypeT errorReportType;
    SaAmfProbableCauseT probableCause;
    SaAmfErrorImpactAndSeverityT errorImpactAndSeverity;
    SaAmfRecommendedRecoveryT recommendedRecovery;
} SaAmfErrorDescriptorT;


typedef void
(*SaAmfHealthcheckCallbackT)(SaInvocationT invocation,
                             const SaNameT *compName,
                             SaAmfHealthcheckT checkType);

typedef void
(*SaAmfReadinessStateSetCallbackT)(SaInvocationT invocation,
                                   const SaNameT *compName,
                                   SaAmfReadinessStateT
readinessState);

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

typedef void
(*SaAmfCSISetCallbackT)(SaInvocationT invocation,
                        const SaNameT *compName,
                        const SaNameT *csiName, /* compName  =
csiName  in NPF */
                        SaAmfCSIFlagsT csiFlags,
                        SaAmfHAStateT *haState,
                        SaNameT *activeCompName,
                     /* compName  = activeCompName in NPF */
                        SaAmfCSITransitionDescriptorT
transitionDescriptor); /* Not used in NPF set to NULL */

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

typedef void
(*SaAmfProtectionGroupTrackCallbackT)
     (const SaNameT *csiName,
     SaAmfProtectionGroupNotificationT *notificationBuffer,
     SaUint32T numberOfItems,
     SaUint32T numberOfMembers,
     SaErrorT error);



typedef void
(*SaAmfPendingOperationConfirmCallbackT)
 (const SaInvocationT invocation,
  const SaNameT *compName,
  SaAmfPendingOperationFlagsT pendingOperationFlags);


typedef struct {
    SaAmfHealthcheckCallbackT
        saAmfHealthcheckCallback;
    SaAmfReadinessStateSetCallbackT
        saAmfReadinessStateSetCallback;
    SaAmfComponentTerminateCallbackT
        saAmfComponentTerminateCallback;
    SaAmfCSISetCallbackT
        saAmfCSISetCallback;
    SaAmfCSIRemoveCallbackT
        saAmfCSIRemoveCallback;
    SaAmfProtectionGroupTrackCallbackT
        saAmfProtectionGroupTrackCallback;
    SaAmfPendingOperationConfirmCallbackT
        saAmfPendingOperationConfirmCallback;
} SaAmfCallbacksT;

    SaErrorT
SaAmfInitialize
(SaAmfHandleT *amfHandle,
 const SaAmfCallbacksT *amfCallbacks,
 const SaVersionT *version);

    SaErrorT
saAmfSelectionObjectGet(const SaAmfHandleT *amfHandle,
                        SaSelectionObjectT *selectionObject);
    SaErrorT
saAmfDispatch(const SaAmfHandleT *amfHandle,
 SaDispatchFlagsT dispatchFlags);

    SaErrorT
saAmfFinalize(const SaAmfHandleT *amfHandle);
    SaErrorT

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

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

    SaErrorT
saAmfCompNameGet(const SaAmfHandleT *amfHandle, SaNameT
*compName);

    SaErrorT
saAmfReadinessStateGet(const SaNameT *compName,
                       SaAmfReadinessStateT *readinessState);

    SaErrorT
saAmfStoppingComplete(SaInvocationT invocation, SaErrorT
error);

    SaErrorT
saAmfHAStateGet(const SaNameT *compName,
 const SaNameT *csiName,
 SaAmfHAStateT *haState);


    SaErrorT
saAmfErrorReport(const SaNameT *reportingComponent,
                 const SaNameT *erroneousComponent,
                 SaTimeT errorDetectionTime,
                 const SaAmfErrorDescriptorT *errorDescriptor,
                 const SaAmfAdditionalDataT *additionalData);

    SaErrorT
saAmfErrorCancelAll(const SaNameT *compName);

    SaErrorT
saAmfComponentCapabilityModelGet(const SaNameT *compName,
                                 SaAmfComponentCapabilityModelT

*componentCapabilityModel);
```

```
    SaErrorT
saAmfPendingOperationGet(const SaNameT *compName,
 SaAmfPendingOperationFlagsT *pendingOperationFlags);

    SaErrorT
saAmfResponse(SaInvocationT invocation, SaErrorT error);


/* checkpoint HA Service data types and prototype */

typedef OPAQUE_TYPE SaCkptHandleT;
typedef OPAQUE_TYPE SaCkptCheckpointHandleT;
typedef OPAQUE_TYPE SaCkptSectionIteratorT;

#define SA_CKPT_WR_ALL_REPLICAS        0X1
#define SA_CKPT_WR_ACTIVE_REPLICA      0X2
#define SA_CKPT_WR_ACTIVE_REPLICA_WEAK 0X4

typedef SaUint32T SaCkptCheckpointCreationFlagsT;

typedef struct {
    SaCkptCheckpointCreationFlagsT creationFlags;
    SaSizeT checkpointSize;
    SaTimeT retentionDuration;
    SaUint32T maxSections;
    SaSizeT maxSectionSize;
    SaUint32T maxSectionIdSize;
} SaCkptCheckpointCreationAttributesT;

#define SA_CKPT_CHECKPOINT_READ      0X1
#define SA_CKPT_CHECKPOINT_WRITE     0X2
#define SA_CKPT_CHECKPOINT_COLOCATED 0X4
typedef SaUint32T SaCkptCheckpointOpenFlagsT;

#define SA_CKPT_DEFAULT_SECTION_ID   {NULL, 0}
#define SA_CKPT_GENERATED_SECTION_ID {NULL, 0}

typedef struct {
    SaUint8T *id;
    SaUint32T idLen;
} SaCkptSectionIdT;

typedef struct {
    SaCkptSectionIdT *sectionId;
    SaTimeT expirationTime;
} SaCkptSectionCreationAttributesT;

typedef enum {
    SA_CKPT_SECTION_VALID = 1,
    SA_CKPT_SECTION_CORRUPTED = 2
} SaCkptSectionStateT;

typedef struct {
    SaCkptSectionIdT sectionId;
    SaTimeT expirationTime;
    SaSizeT sectionSize;
    SaCkptSectionStateT sectionState;
    SaTimeT lastUpdate;
} SaCkptSectionDescriptorT;

typedef enum {
    SA_CKPT_SECTIONS_FOREVER = 1,
    SA_CKPT_SECTIONS_LEQ_EXPIRATION_TIME = 2,
    SA_CKPT_SECTIONS_GEQ_EXPIRATION_TIME = 3,
    SA_CKPT_SECTIONS_CORRUPTED = 4,
    SA_CKPT_SECTIONS_ANY = 5
} SaCkptSectionsChosenT;

typedef struct {
    SaCkptSectionIdT sectionId;
    void *dataBuffer;
    SaSizeT dataSize;
    SaOffsetT dataOffset;
    SaSizeT readSize;
} SaCkptIOVectorElementT;


typedef struct {
    SaCkptCheckpointCreationAttributesT
checkpointCreationAttributes;
    SaUint32T numberOfSections;
    SaUint32T memoryUsed;
} SaCkptCheckpointStatusT;


typedef void
(*SaCkptCheckpointOpenCallbackT)(SaInvocationT invocation,
const SaCkptCheckpointHandleT
*checkpointHandle,
SaErrorT error);

typedef void
(*SaCkptCheckpointSynchronizeCallbackT)
(SaInvocationT invocation, SaErrorT error);

typedef struct {
    SaCkptCheckpointOpenCallbackT saCkptCheckpointOpenCallback;
    SaCkptCheckpointSynchronizeCallbackT
saCkptCheckpointSynchronizeCallback;
} SaCkptCallbacksT;

    SaErrorT
SaCkptInitialize
(SaCkptHandleT *ckptHandle,
 const SaCkptCallbacksT *callbacks,
 const SaVersionT *version);

    SaErrorT
saCkptSelectionObjectGet(const SaCkptHandleT *ckptHandle,
                         SaSelectionObjectT *selectionObject);

    SaErrorT
saCkptDispatch(const SaCkptHandleT *ckptHandle,
               SaDispatchFlagsT dispatchFlags);

    SaErrorT
saCkptFinalize(const SaCkptHandleT *ckptHandle);

    SaErrorT
saCkptCheckpointOpen(const SaNameT *ckeckpointName,
const SaCkptCheckpointCreationAttributesT
*checkpointCreationAttributes,
SaCkptCheckpointOpenFlagsT checkpointOpenFlags,
SaTimeT timeout,
SaCkptCheckpointHandleT *checkpointHandle);
```

```
    SaErrorT
saCkptCheckpointOpenAsync(const SaCkptHandleT *ckptHandle,
SaInvocationT invocation,
const SaNameT *ckeckpointName,
const SaCkptCheckpointCreationAttributesT
*checkpointCreationAttributes,
SaCkptCheckpointOpenFlagsT checkpointOpenFlags);

    SaErrorT
saCkptCheckpointClose(const SaCkptCheckpointHandleT
*checkpointHandle);

    SaErrorT
saCkptCheckpointRetentionDurationSet(const
SaCkptCheckpointHandleT *checkpointHandle,
SaTimeT retentionDuration);

    SaErrorT
saCkptActiveCheckpointSet(const SaCkptCheckpointHandleT
*checkpointHandle);

    SaErrorT
SaCkptCheckpointStatusGet
(const SaCkptCheckpointHandleT *checkpointHandle,
SaCkptCheckpointStatusT *checkpointStatus);

    SaErrorT
SaCkptSectionCreate
(const SaCkptCheckpointHandleT *checkpointHandle,
SaCkptSectionCreationAttributesT *sectionCreationAttributes,
const void *initialData,
SaUint32T initialDataSize);

    SaErrorT
SaCkptSectionDelete
(const SaCkptCheckpointHandleT *checkpointHandle,
const SaCkptSectionIdT *sectionId);


    SaErrorT
SaCkptSectionExpirationTimeSet
(const SaCkptCheckpointHandleT *checkpointHandle,
const SaCkptSectionIdT* sectionId,
SaTimeT expirationTime);

    SaErrorT
SaCkptSectionIteratorInitialize
(const SaCkptCheckpointHandleT *checkpointHandle,
SaCkptSectionsChosenT sectionsChosen,
SaTimeT expirationTime,
SaCkptSectionIteratorT *sectionIterator);

    SaErrorT
SaCkptSectionIteratorNext
(SaCkptSectionIteratorT *sectionIterator,
 SaCkptSectionDescriptorT *sectionDescriptor);

    SaErrorT
SaCkptSectionIteratorFinalize
(SaCkptSectionIteratorT *sectionIterator);

    SaErrorT
SaCkptCheckpointWrite
(const SaCkptCheckpointHandleT *checkpointHandle,
const SaCkptIOVectorElementT *ioVector,
SaUint32T numberOfElements,
SaUint32T *erroneousVectorIndex);
```

```
    SaErrorT
SaCkptSectionOverwrite
(const SaCkptCheckpointHandleT *checkpointHandle,
const SaCkptSectionIdT *sectionId,
SaUint8T *dataBuffer,
SaSizeT dataSize);

    SaErrorT
SaCkptCheckpointRead
(const SaCkptCheckpointHandleT *checkpointHandle,
SaCkptIOVectorElementT *ioVector,
SaUint32T numberOfElements,
SaUint32T *erroneousVectorIndex);

    SaErrorT
SaCkptCheckpointSynchronize
(const SaCkptCheckpointHandleT *ckeckpointHandle,
SaTimeT timeout);

    SaErrorT
SaCkptCheckpointSynchronizeAsync
(const SaCkptHandleT *ckptHandle,
SaInvocationT invocation,
const SaCkptCheckpointHandleT *checkpointHandle);


/* Event Service API data types and prototypes */


typedef OPAQUE_TYPE SaEvtHandleT;
typedef OPAQUE_TYPE SaEvtEventHandleT;
typedef OPAQUE_TYPE SaEvtChannelHandleT;
typedef SaUint32T SaEvtSubscriptionIdT;


typedef void
(*SaEvtEventDeliverCallbackT)
(const SaEvtChannelHandleT *channelHandle,
SaEvtSubscriptionIdT subscriptionId,
const SaEvtEventHandleT *eventHandle,
const SaSizeT eventDataSize);

typedef struct{
    const SaEvtEventDeliverCallbackT saEvtEventDeliverCallback;
} SaEvtCallbacksT;

#define SA_EVT_CHANNEL_PUBLISHER  0X1
#define SA_EVT_CHANNEL_SUBSCRIBER 0X2
#define SA_EVT_CHANNEL_CREATE     0X4
typedef SaUint8T SaEvtChannelOpenFlagsT;

typedef struct {
    SaUint8T *pattern;
    SaSizeT patternSize;
} SaEvtEventPatternT;


#define SA_EVT_HIGHEST_PRIORITY 0
#define SA_EVT_LOWEST_PRIORITY  3

#define SA_EVT_LOST_EVENT "SA_EVT_LOST_EVENT_PATTERN"

typedef struct {
    SaEvtEventPatternT *patterns;
    SaSizeT patternsNumber;
} SaEvtEventPatternArrayT;
```

```
typedef SaUint8T SaEvtEventPriorityT;
typedef SaUint32T SaEvtEventIdT;

typedef enum {
    SA_EVT_PREFIX_FILTER = 1,
    SA_EVT_SUFFIX_FILTER = 2,
    SA_EVT_EXACT_FILTER = 3,
    SA_EVT_PASS_ALL_FILTER = 4
} SaEvtEventFilterTypeT;

typedef struct {
    SaEvtEventFilterTypeT filterType;
    SaEvtEventPatternT filter;
} SaEvtEventFilterT;

typedef struct {
    SaEvtEventFilterT *filters;
    SaSizeT filtersNumber;
} SaEvtEventFilterArrayT;

    SaErrorT
SaEvtInitialize
(SaEvtHandleT *evtHandle, const SaEvtCallbacksT *callbacks,
 const SaVersionT *version);

    SaErrorT
saEvtSelectionObjectGet(const SaEvtHandleT *evtHandle,
                        SaSelectionObjectT *selectionObject);

    SaErrorT
SaEvtDispatch
(const SaEvtHandleT *evtHandle,
SaDispatchFlagsT dispatchFlags);

    SaErrorT
saEvtFinalize(SaEvtHandleT *evtHandle);

    SaErrorT
saEvtChannelOpen(const SaEvtHandleT *evtHandle,
const SaNameT *channelName,
SaEvtChannelOpenFlagsT channelOpenFlags,
SaEvtChannelHandleT *channelHandle);

    SaErrorT
saEvtChannelClose(SaEvtChannelHandleT *channelHandle);

    SaErrorT
SaEvtEventAttributesSet
(const SaEvtEventHandleT *eventHandle,
 const SaEvtEventPatternArrayT *patternArray,
 SaUint8T priority,
 SaTimeT retentionTime,
 const SaNameT *publisherName);

    SaErrorT
SaEvtEventAttributesGet
(const SaEvtChannelHandleT *channelHandle,
const SaEvtEventHandleT *eventHandle,
SaEvtEventPatternArrayT *patternArray,
SaUint8T *priority,
SaTimeT *retentionTime,
SaNameT *publisherName,
SaClmNodeIdT *publisherNodeId,
SaTimeT *publishTime,
SaEvtEventIdT *eventId);
```

```
    SaErrorT
saEvtEventDataGet(const SaEvtEventHandleT *eventHandle,
                  void *eventData,
                  SaSizeT *eventDataSize);

    SaErrorT
saEvtEventPublish(const SaEvtChannelHandleT *channelHandle,
                  const SaEvtEventHandleT *eventHandle,
                  const void *eventData,
                  SaSizeT eventDataSize);

    SaErrorT
saEvtEventSubscribe(const SaEvtChannelHandleT *channelHandle,
                    const SaEvtEventFilterArrayT *filters,
                    SaEvtSubscriptionIdT subscriptionId);

    SaErrorT
saEvtEventUnsubscribe(const SaEvtChannelHandleT *channelHandle,
                      SaEvtSubscriptionIdT subscriptionId);

    SaErrorT
SaEvtEventRetentionTimeClear
(const SaEvtChannelHandleT *channelHandle,
const SaEvtEventHandleT *eventHandle);
```

APPENDIX A ACKNOWLEDGEMENTS

Working Group Chair: Vinoj Kumar

Task Group Chair: Ram Gopal. L

The following individuals are acknowledged for their participation to High Availability task group teleconferences, plenary meetings, mailing list, and/or for their NPF contributions used for the development of this Implementation Agreement. This list may not be all-inclusive since only names supplied by member companies for inclusion here will be listed. The NPF wishes to thank all active participants to this Implementation Agreement, whether listed here or not.

The list is in alphabetical order of last names:

Bachmutsky Alex(Nokia)

Balakrishnan Santosh (Intel)

Damon Pilippe (IBM)

Keany Berny(Intel)

Keisu Torbjorn (Ericsson)

Kumar Vinoj(Agere)

Lewing Van (PMC-Sierra)

Muralidhar Rajeev (Intel)

Papp William (Nokia)

Renwick John (Agere)

Sam (Intel)

Sridhar T(FutureSoft)

Stone Alan (Intel)

Vandalore Bobby (Nokia)

Vedvyas Shanbhogue (Intel)

Verma Sanjeev(Nokia)

APPENDIX B LIST OF COMPANIES BELONGING TO NPF DURING APPROVAL PROCESS

Agere Systems	FutureSoft	Nokia
Cypress Semiconductor	Intel	PMC Sierra
Ericsson	IP Infusion	Sun Microsystems
ETRI	Kawasaki LSI	Xilinx