

Mapping Java EE to SAForum Specifications: A NEP Perspective

Version 1.0, 2007-10-11

Copyright © 2007 SCOPE Alliance. All rights reserved.

The material contained herein is not a license, either expressed or implied, to any IPR owned or controlled by any of the authors or developers of this material or the SCOPE Alliance. The material contained herein is provided on an “AS IS” basis and to the maximum extent permitted by applicable law, this material is provided AS IS AND WITH ALL FAULTS, and the authors and developers of this material and SCOPE Alliance and its members hereby disclaim all warranties and conditions, either expressed, implied or statutory, including, but not limited to, any (if any) implied warranties that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

Also, there is no warranty or condition of title, quiet enjoyment, quiet possession, correspondence to description or non-infringement with regard to this material. In no event will any author or developer of this material or SCOPE Alliance be liable to any other party for the cost of procuring substitute goods or services, lost profits, loss of use, loss of data, or any incidental, consequential, direct, indirect, or special damages whether under contract, tort, warranty, or otherwise, arising in any way out of this or any other agreement relating to this material, whether or not such party had advance notice of the possibility of such damages.

Questions pertaining to this document, or the terms or conditions of its provision, should be addressed to:

SCOPE Alliance,
c/o IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08854
Attn: Board Chairman

Or

For questions or feedback, use the web-based forms found under the Contacts tab on www.scope-alliance.org

1. PURPOSE

The SCOPE alliance was established early in 2006 to foster the growth of the carrier grade base platform COTS supply chain ecosystem. The carrier grade base platform includes Carrier Grade hardware, operating systems, and middleware.

NGN (Next Generation Networks) and Service layer applications have high requirements on rapid development and deployment of new services. To meet these requirements Java EE is used to a large extent, however in the telecom domain, carrier grade high availability and fault tolerance is a norm and it also needs to be valid for Java EE applications to a larger extent than today.

The most important problem to solve is to provide high availability to Java EE applications without the need for them to adapt to a new framework. It is important to leverage any modifications on the Carrier-Grade Base Platform Middleware as defined in the SCOPE Alliance reference architecture [3] and in SA Forum documentation [1] as the foundation for the Java EE HA environment. This document will show how Java EE can be integrated with a standards based SAF middleware and provide high availability to Java EE applications without putting unnecessary burden on the application developer.

For the sake of clarity, the terms “middleware,” “service,” “component” and “application” as used in this document will, to the extent possible, be consistent with the SA Forum usage of the terms:

- **Middleware** – carrier-grade base platform middleware as defined in [3]. This contains platform-level services such as support for service availability, but not application services such as databases or application servers.
- **Component** - represents a set of resources including hardware and software resources. It is realized by operating system processes. [1]
- **Application** – Aggregates components to provide a higher level of service [1]. Java EE applications run in an application server and from an SAF perspective the application server and the Java EE application together is a SAF application.
- **Service** –refers to the SAF high-level services (CLM, CKPT, EVT, MSG, LCK, NTF, LOG, and IMM) [5]. More generally, a *service* is a set of actions that satisfy a request from a user or other system. For example, database servers satisfy requests for updates, inserts, or queries. Services may embed other subsidiary services. For example, a database-driven Web service will invoke database servers. [4]
- **Availability** in general refers to the probability that a system is in an operating condition that allows it to provide the service(s) for which it is intended to perform. Availability is a measure that conventionally is expressed as a percentage which is calculated as the uptime divided by the uptime plus the downtime of the system. For this consideration it makes no difference whether the downtime is planned or unplanned. Often availability is expressed as the number of nines (e.g., 99.999% is expressed as five 9's and this translates to approximately 5.26 minutes of downtime per year). [4]

The baseline for this profiling is the SA Forum Release 2007 version B.03.01 from February 2007 [5].

2. AUDIENCE

This document is intended for the following audiences:

- Developers of carrier-grade service-availability middleware services
- Third-party component developers, suppliers, and consumers
- Developers, suppliers, and consumers of integrated product platforms that include the service availability layer
- Developers of service availability interface specifications.
- The Java community, developers of Application servers and applications.

3. REFERENCES

1. AIS Tutorial. Available at http://www.saforum.org/press/presentations/AIS_Tutorial_final_Nov.pdf
2. SCOPE Carrier Grade Middleware profile Version 1.0, February 8, 2007 available at http://www.scope-alliance.org/pr/SCOPE_CG_Middleware_profile_v1.0.pdf and SCOPE Carrier Grade Middleware Profile Version 2.0, July 16, 2007 available at http://www.scope-alliance.org/pr/CG_MW_Profile_07_07_16_v2.0.pdf
3. SCOPE Technical Position Paper: Scoping the SCOPE – Closing the Gaps of Open Carrier Grade Base Platforms, Version 1.1, Available at <http://www.scope-alliance.org/scope-technical-position.pdf>
4. Services and Support Profile – Service Availability, July 30, 2007 available at http://www.scope-alliance.org/docs/Services-Profile_Service-Availability_v1.0.pdf
5. Service Availability Forum Service Availability Interface Overview (SAI-Overview-B.03.01).
6. Service Availability Forum Hardware Platform Interface (SAI-HPI-B.02.01)..
7. Service Availability Forum Application Interface Specification Checkpoint Service (SAI-AIS-CKPT-B.02.01).
8. Service Availability Forum HPI-to-AdvancedTCA Mapping Specification (SAIM-HPI-B.01.01-ATCA). December 20, 2005.
9. Java Message Service <http://java.sun.com/products/jms/docs.html>
10. Java Management Extension, JMX <http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html>

11. Java Logging API
<http://java.sun.com/javase/6/docs/technotes/guides/logging/index.html>
12. Java Preferences API
<http://java.sun.com/j2se/1.5.0/docs/guide/preferences/index.html>
13. Java Persistence API
<http://java.sun.com/developer/technicalArticles/J2EE/jpa/>
14. Java Naming and Directory Interface
<http://java.sun.com/products/jndi/tutorial/>

All of the Service Availability Forum documents are available at their web site
<http://www.saforum.org>.

4. INTRODUCTION

The availability support that exists for Java EE today covers only Java parts and does not include availability support for hardware and native applications. It is desirable to have a framework that can provide availability for both Java and native applications in the same clustered system (see Fig. 1). One important thing to point out is that the framework should not impose any native programming style on the Java parts and it should be possible to run Java applications that are unaware of the framework. Another important part is that in a highly available clustered system only one entity can take the decisions and control the availability.

The same problem exists for configuration and fault management when Java and native application should be hosted in the same cluster. It is important that configuration and fault management is aligned between native and Java applications. One way to solve this problem is to use a standardized framework for availability and configuration and fault management as the base for both Java and native applications.

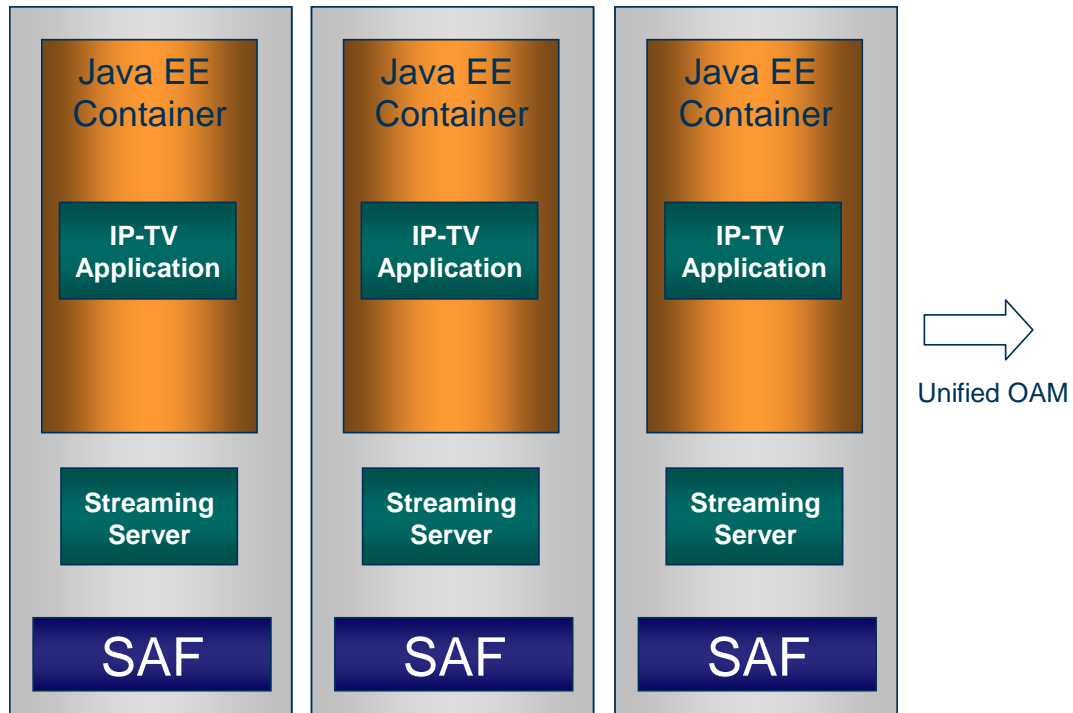


Fig 1: Java and native applications can be combined to provide one service. The native streaming server and they Java application are both controlled by the same middleware and have a unified OAM.

5. TERMS AND DEFINITIONS

AIS	Application Interface Specification (SA Forum term)
AMF	Availability Management Framework (SA Forum term)
CKPT	Checkpoint Service (SA Forum term)
CLM	Cluster Membership Service (SA Forum term)
EJB	Enterprise Java Beans
EVT	Event Service (SA Forum term)
IMM	Information Model Management Service (SA Forum term)
ISV	Independent Software Vendor
AMJ	Availability Management for Java
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JDO	Java Data Object
JMX	Java Management Extension
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JSR	Java Specification Request
MSG	Message Service (SA Forum term)
NTF	Notification Service (SA Forum term)
NEP	Network Equipment Provider

OAM	Operation and Maintenance
-----	---------------------------

6. HIGH AVAILABILITY FOR JAVA EE

Today different Java EE implementations have different levels of availability support some have more and some less. One important thing to notice is that they are all different and there does not exist any standardized way to solve the availability problem for Java EE. The availability support that exists for Java EE today does not cover the complete system it only covers the Java parts. This means that another framework is needed for handling of native applications and things that are not covered, i.e. HW Management etc. The same problem exists for OAM where the existing Java EE implementations do not have sufficient support for fault management and configuration management. It is important that the OAM for Java EE applications, native applications and the platform is integrated.

It is important to get a standardized way of handling availability for Java EE. The standard should make it possible to use different availability frameworks as the base for the Java EE availability and specifically the Carrier-Grade Base Platform Middleware as defined in the SCOPE Alliance reference architecture [3]. It should also be possible to change between different availability frameworks without affecting the Java EE applications.

The standard should not enforce any changes to existing Java EE applications but it should give the possibility for new applications to make use the availability support. In this way all Java EE applications can take advantage of the availability support and applications that need the functionality provided by the availability API can use it. It should also be possible for Java EE applications to inject classes that are aware of the availability framework and in this way make it possible to decouple the availability awareness and the Java EE application.

The following are important availability areas that need to be addressed for Java EE. (The solution should not enforce changes to existing Java EE applications):

- Redundancy for application components and containers.
This is handled in different ways by many Java EE implementations and it only covers the Java parts. An availability framework that can handle both Java and native applications is preferred.
- Fault Management.
Fault management needs follow the ITU-T X.700-X.799 recommendations. Integration between fault management for native applications and Java EE applications is needed. The integration should be done in a way that Java applications still can use JMX for fault management.
- Configuration Management
In the same way as for fault management the configuration management needs to be integrated for native applications and Java applications.
- Software Upgrade
In service software upgrade is needed for all telecom system and it needs to be able to handle both native and Java EE applications.

7. JAVA EE AND SAF

Not all of the services defined in the SAF standard are of interest for Java EE. For some of the services there already exist the same or similar functionality standardized for Java and there is no need for an interaction between Java and native applications. For other SAF services there are many of reasons for integration.

- **OAM Alignment**
Java applications and native applications will be hosted in the same environment. One example of native applications that already are co-located with Java is data-bases. It is strongly desirable that both types of applications can be managed the same way.
- **Portability**
It must be possible to move containers between different HA middle wares and it must be possible to move applications from one container to another.
- **SAF Enabled Third party products**
When the eco system around SAF takes of there will be a lot of SAF enabled 3pp products available. It is desirable that these products can be used together with Java EE as well.
- **High Availability**
When Java EE is to be used in a mixed environment with native applications it is important that one framework controls the both Java and native application redundancy and distribution. A new Java API is needed for this; the working name for it is Availability Management API for Java.

For some of the SAF services there is no need for integration with Java. If there already exist an accepted standard within the Java community and there is no need for interaction between Java and native applications the integration is unnecessary. One example is the checkpoint service where there already exist alternatives within the Java community and there is no need for Java and native applications to share a checkpointed state.

7.1 Availability Management Framework

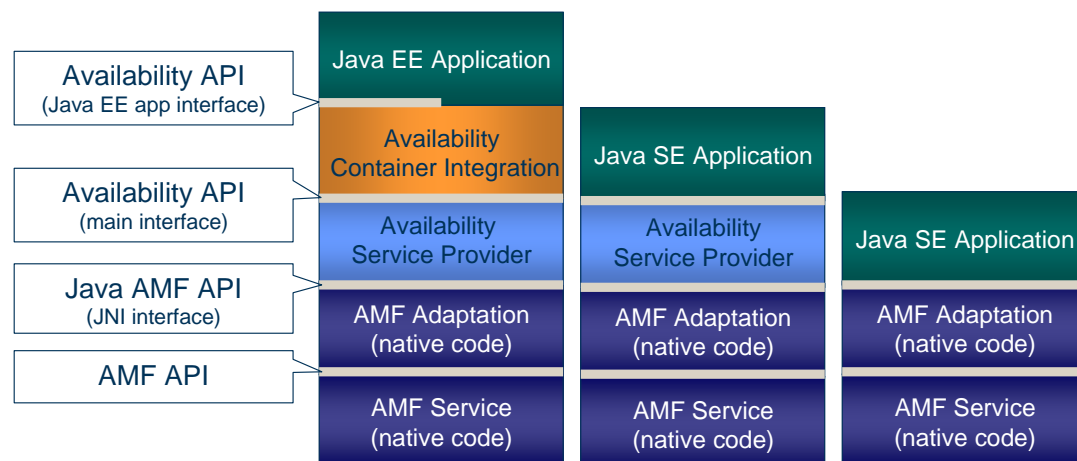


Fig 2: Layered architecture of integration with AMF and different ways to use it.

The integration with AMF from Java is done in different layers. First a low level Java API for AMF is needed (“Java AMF API (JNI interface) in Fig. 2). This API will be more or less a direct mapping of the AMF functionality described by SAF. This API can then be used by the service provider for implementation of the Availability Management API or even by Java SE applications directly. The service provider implementation (“Availability API (main interface)” in Fig. 2) is then used by the container implementation of the Availability Management for Java that in its turn provides a simplified and optional interface (“Availability API (Java EE app interface)” in Fig. 2) to the Java EE applications. It is possible to have different service providers for different middlewares.

An Availability Management API that makes it possible to integrate with existing HA middle wares should have the following goals:

- It shall not specify the availability framework itself but it shall only specify the means by which the framework can supervise and control the Java units within a JVM. The means by which the framework instantiates JVMs is outside the scope of the specification.
- It shall allow different service providers to provide support for specific availability frameworks, standardized or proprietary. It is required that an implementation for the standardized AMF (Availability Management Framework) of SA Forum shall be feasible.
- It shall be designed with Java EE as the main target, although parts will also be useful on Java SE. The constraints set by the component a model of Java EE needs to be considered.
- It shall specify a basic set of features that can be considered as useful for Java EE and possible to support by most availability frameworks. This implies that only a subset of the features of AMF will be supported.
- It shall support Java EE applications that are unaware, partially aware or completely aware of the control of the availability framework. It is anticipated that the main part of the API is implemented in the Java EE server and that existing Java EE applications can take advantage of the availability support without any changes.
- It shall not handle all aspects of clustered Java systems. Especially it shall not specify any state replication solution, although it may specify that the API can give hints to such replication solutions in the form of reasons for the activation or the deactivation of a unit.

7.2 Cluster Membership

The SAF Cluster Membership Service (CLM) is a candidate for Java standardization. There is no overlapping functionality standardized in Java but there exist some open source implementations with similar functionality.

- JGroups is an open source toolkit from JBoss for reliable multicast communication.
- JXTA is an open source peer-to-peer protocol from Sun Microsystems. It allows any device connected to a network to exchange messages and collaborate independently of the underlying network topology.

One of these or other similar implementations could be used as an alternative to the Cluster Membership but it is not standardized. A standardized version of CLM could be the base for many Java services that need to know about the cluster. There should be a single, platform-wide clustering service that serves native and Java applications (vs. independent clustering mechanisms using different policies and/or making separate decisions). This implies that the Java interface implementation be derived from the native CLM service on the platform.

7.3 Checkpoint Service

The Java community has standardized alternatives to the SAF Checkpoint Service (CKPT) and there are alternative solutions in the Open Source Community.

One of the alternatives is the Java Persistence API, JSR 220 [13]. The Java Persistence API is a lightweight framework for persisting Java objects. It is considered the way forward for both J2EE and J2SE applications. Even if it is mainly designed to use a database for storing the data, it can be extended to protect the data in other ways (i.e. in memory replication). The Persistence API combines the best ideas from other popular persistence frameworks such as Hibernate, JDO, TopLink and others.

JBoss Cache is an open source implementation that is not standardized but it solves the same problem as the SAF Checkpoint Service. It works as a distributed cache for Java objects with in memory replication. It has 3 different caching modes:

- Local Cache – without replication
- Replicated Cache – using non-blocking asynchronous replication.
- Replicated Cache – using blocking synchronous replication.

It also has transactional support and pluggable policies for database integration.

As there is no need for native and Java applications to share a check-pointed state and there are solutions within the Java community that covers check-pointing the SAF Checkpoint service is not needed for Java EE.

7.4 Event Service

In the same way as for the SAF Checkpoint Service there are alternative solutions to the SAF Event Service (EVT) in the Java Community. Java Message Service (JMS) [9] has more or less the same functionality as the Event Service from SAF.

Java Message Service provides two types of messaging models, publish-subscribe and point-to-point queuing. In publish-subscribe one producer can send a message to many consumers through a channel. Subscribers can choose to subscribe to a specific topic and any message addressed to that topic is sent to all consumers that have subscribed to it. The publisher is not dependent on the consumer(s) that receive the message. JMS publish-subscribe is very similar to the SAF Event Service and there is no gain in introducing something new in the Java Community that does not provide any new functionality. To conclude the Event Service is probably not a candidate for use in Java EE but if Java applications and native C/C++ applications in a cluster need to communicate, integration between JMS and the Event Service could be done.

7.5 Message Service

The Java Message service also contains one model that has overlapping functionality with the SAF Message Service (MSG).

The JMS point-to-point messaging model allows JMS clients to send and receive messages via queues. A given queue may have multiple receivers but only one receiver may consume each message. JMS takes care of distributing the messages within a group of receivers and guarantees that the message is delivered once and only once.

In the same way as for the Event Service, there already exists standardized functionality in Java that is similar so there is no need to use the SAF service from Java. If Java applications and native C/C++ applications in a cluster need to communicate, integration between JMS and the Event Service could be done.

It has not been investigated if JMS is good enough for all types of telco applications that should run in a Java environment.

7.6 Log Service

Java has a logging API [11] so a new logging API based on SAF logging service (LOG) should not be defined. The log handling needs to be integrated. All logs produced by applications in a cluster should be handled the same way.

The integration can be done with a Log Handler that will send log records to the SAF service. As the Java logging service is a mix of a trace and log service it is probable that not all log records should be forwarded, rather only log records that have a higher log level. The Java logs will be sent to a system or application log stream in the SAF Log. There is no need for the alarm and notification logs from Java, this can be done through the NTF-JMX integration.

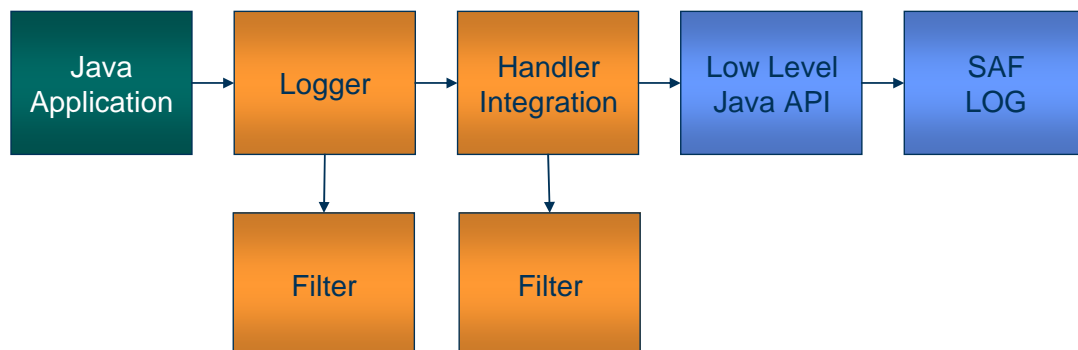


Fig 3: Overview of the integration between the SAF Log service and Java logging.

7.7 Naming Service

The Java Naming and Directory Interface (JNDI) is a naming and directory API for Java applications. The architecture of JNDI includes a service provider interface which enables a variety of naming and directory services to be used by the applications via the JNDI API.

The SAF Naming service can be integrated with JNDI to provide a common naming service for native and Java applications.

7.8 Timer Service

In Java there exist different timers. For example web applications, EJB applications, Swing applications and SE applications have their own timers.

As there already exist many different timer APIs in Java and timers are local to the applications there is no need for integration between any of the Java timers and the SAF timer service.

7.9 Notification Service

The functionality in the SAF Notification Service (NTF) is partly covered by JMX [10] in Java.

The Java Management Extension provides a standard way to enable manageability for Java based applications. It contains smart agents capable of being managed through protocols like SNMP and WEBM.

Integration between JMX and the Notification Service will provide the Java EE environment with an ITU-T X.700-X.799 based fault management system. It will also give a uniform handling of alarms and notifications.

The Java applications are instrumented by means of JMX to emit JMX notifications structured in a way that is feasible to map to NTF. It may also be possible to standardize an alarm facility API that facilitates the structuring and the emission of these JMX notifications.

In a system solution supporting both native C/C++ applications and Java applications there must be some interaction between the NTF service and the JMX agent in order to support the following scenarios.

- A notification subscriber is using the Subscriber API of NTF in order to subscribe for notifications from both C/C++ applications and Java applications (via JMX).
- A notification reader is using the Reader API of NTF in order to retrieve historical notification entries from both C/C++ applications and Java applications (via JMX).



Fig 4: Integration between JMX and NTF.

Java applications can use MBeans to send JMX notifications that will be mapped to alarms. Mapping rules are needed to get the right information to the notification service. A JMX Agent can use a low level Java interface to the producer interface of NTF to send the alarms and notifications. In this way all alarms and notifications can be handled in a uniform way for a system with both Java and native applications.

7.10 Information Model Management Service

In some parts JMX and Java Preferences [12] have an overlapping functionality with IMM. Integration between IMM and JMX will give a uniform handling of configuration.

Java applications are instrumented by the means of JMX and uses MBeans to get configuration changes.

In a system solution supporting both C/C++ applications and Java applications there must be some interaction between the IMM service and the JMX agent in order to support the following scenarios.

- A system management application uses the Object Management API of IMM and the IMM service provides the configuration support for both C/C++ applications and Java applications (via JMX).
- A system management application uses the JMX interfaces and the JMX agent provides the configuration support for both C/C++ applications (via IMM) and Java applications.

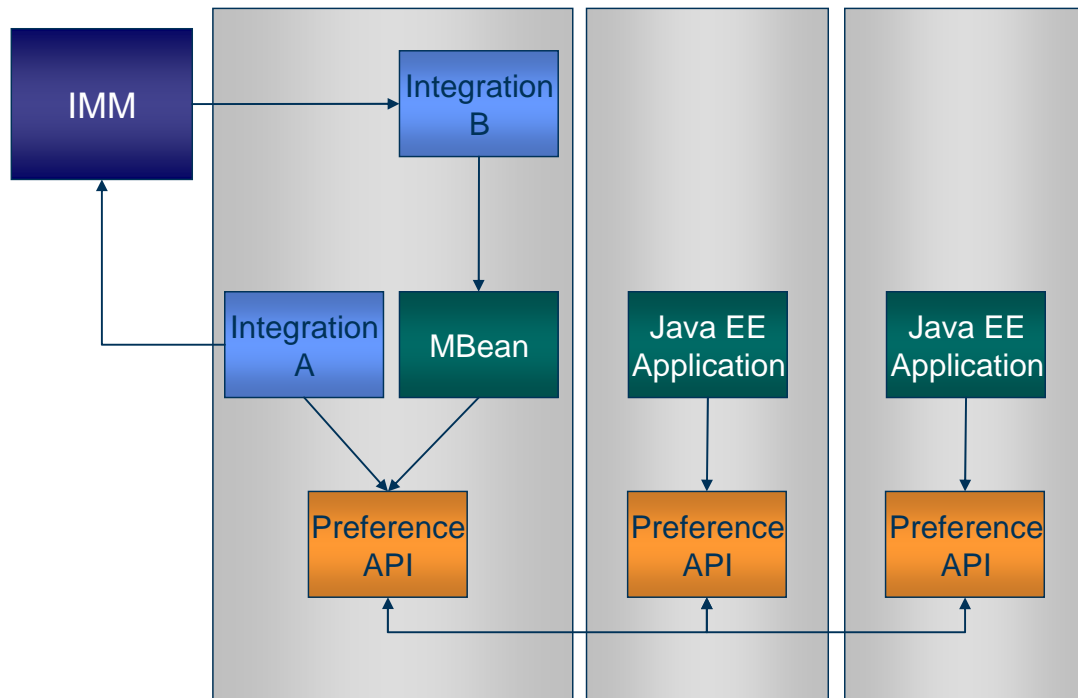


Fig 5 : Integration between IMM, JMX and Preference API

The initial configuration is fetched from IMM by integration implementation A in Fig 5 above and the Java Preference Framework is used to make the configuration available to Java EE applications on all nodes in a cluster. When the configuration data is updated in IMM integration implementation B will use JMX to locate the appropriate configuration MBean and forward the updated data to it. The configuration MBean will validate the data and write it to the Java Preferences Framework. The Java Preference Framework should have a backend storage that distributes the data to all nodes in the cluster. When the data is updated the Java Preference Framework will notify the applications and they can read the data.

A new Java API for distribution of configuration in a clustered environment should be considered for future standardization. It is also possible to replace the Preference API in the integration with JMX only.

8. CONCLUSION

Java EE can be integrated with the Carrier-Grade Base Platform Middleware and provide an environment that makes it possible to combine Java and native applications in the same cluster. It is important that Java applications have the same level of availability as native applications and this can be achieved with a new Java API, Java Availability API, which makes it possible to integrate with AMF.

It is equally important to get an alignment of operation and maintenance between Java and native applications. For these areas Java has standardized APIs and integration with SAF is needed. The most important integrations are:

- NTF integration with JMX
- IMM integration with JMX and Java Preferences API
- LOG integration with Java logging