This report summarizes our comparative study between the CCSDS Asynchronous Message Service (AMS) and the Java Message Service (JMS).

1. Introduction.

The purpose of AMS is to allow communications between modules of a mission system. Each module may be designed to operate in isolation, producing and consuming information without explicit awareness of other operating modules. AMS provides a standard infrastructure for the exchange of information in a flexible, robust, scalable and efficient way.

Enterprise messaging products (i.e. Message Oriented Middleware products) are becoming an essential component for integrating intra and inter company operations. They allow separate components to be combined into a reliable and flexible system. JMS provides a common way for Java language programs to interact with these systems. It defines a set of interfaces and associated semantics that define how a JMS client uses the facilities of an enterprise messaging product.

AMS and MOM/JMS have a set of common objectives: design and implementation of component based applications in the context of loosely coupled distributed systems. However their initial targets are quite different; while JMS is designed for application integration in multi-organization environments over the Internet (EAI, B2B solutions), AMS targets embedded systems through proprietary networks.

Last, the level of the two specifications are not really the same; while AMS describes a full featured data system communications architecture, JMS just specifies the client API that allows Java programs to communicate through an asynchronous communication system. As a consequence, the architecture part of the JMS specification is reduced to the client/provider link. In order to extend the comparison with AMS, we will study a provider architecture with Joram.

JORAM (Java Open Reliable Asynchronous Messaging) is an open source implementation of the JMS™ messaging specification. JORAM is fully compliant with JMS 1.1 and is available on a large range of computer systems from application servers to internet appliances.

This document provides a general comparison of the CCSDS Asynchronous Message Service (AMS) and the Joram JMS (Java Message Service) implementation. Section 2 is a brief introduction to AMS. Section 3 describes the JMS specification and its Joram implementation. It presents the JORAM architecture and highlights its distributed and configurable architecture. Section 4 describes the similarities and differences between
AMS and JMS/Joram. Finally section 5 discusses about the way Joram can be used for the construction of an AMS platform.

2. AMS overview

The CCSDS Asynchronous Message Service\(^1\) is designed to build a data centric application where each module of the application can work in isolation; each module produces and consumes data without explicit awareness of what other modules are currently operating:

- Any module may be introduced into (resp. removed from) the system at any time, the starting order of the different modules is immaterial; modules do not need to synchronize in order to exchange messages.

- The termination of any module does not cause the implicit termination of other modules; a module can be upgraded at any time. The termination of the whole system when needed must be operated in an immaterial order.

Consequently, AMS based system are highly robust and configurable. Additionally, communication within an AMS based system must be rapid and efficient: messages are exchanged directly between modules rather than through a central dispatching node; they are transmitted using the best underlying transport service.

2.1. AMS model

AMS messages are fundamentally asynchronous, an AMS message sender does not need to wait for the message being delivered to continue its function.

Due to its asynchronous behavior it may be desirable to link the information in a reply message to the context of the corresponding request; for this purpose AMS enables to attach a context number in any original message. The AMS interface can be used to reply to any original message, and the reply message automatically includes the related context number.

Sometimes it may be necessary to synchronize the sender with the receiver; for this purpose AMS allows a node to suspend its execution until a reply is received.

Most messages are exchanged in an AMS system according to the Publish/Subscribe paradigm:

- Any node can use the AMS interface to subscribe to messages on a specified subject;

- When a node uses the AMS interface to publish a message on that subject, the system ensures the message delivery to all the nodes that have previously subscribed to this subject.

This model greatly simplifies application development and integration, allowing each node design to be largely uncoupled from the designs of all others. However some usages may still require a node to send a message to an explicitly specified node; the AMS interface also supports this communication model.

---

\(^1\) Cf. “AMS proposed standard” and “AMS – JPL implementation U.G.”
2.2. AMS architecture

A **continuum** is a closed set of entities that utilize AMS for purposes of communication among themselves.

A **node** is a communicating entity that implements a part of the functionality of the AMS **application** - among other activities it exchanges messages with other nodes through the AMS interface.

A **message space** is the set of all nodes of one AMS **application instance** in a single continuum. A **zone** is an administrative subset of the message space declared during the application configuration.

A **role** is some part of the functionality of an application. Each role is identified by a role name and a corresponding non-negative role number.

Each zone is uniquely identified in the message space by its zone name and corresponding zone number. A message space comprises at least one zone, the **root zone**, which contains all nodes of the message space; its zone name is the string that is of length zero, its zone number is zero. A zone whose name is the beginning of the name of another zone of the same message space is said to contain that other zone. The membership of a zone that is contained by another zone is a subset of the membership of the containing zone. The registered membership of a zone is the set of all nodes in the zone that are not members of any other zone which does not contain that zone.

Partitioning message spaces into zones helps AMS to be scalable; it enables an application to work over thousands of nodes without significant impact on application performance.

**Message format**

A message is a byte array of known size. The **subject** of a message is an integer related to the application that indicates the general nature of the information conveyed by the message. The **subject name** is a string that gives a symbolic representation of the subject number.

The structure of the content of messages on a given subject is application specific and is not described by the AMS protocol. However for each message subject used in the message space the AMS MIB will contain:

- The name and integer identifying the message subject.
- A semantic description of the message subject.
- A detailed specification of the structure of the content for messages on this subject.
- An optional specification of the marshalling of this type of messages.

When AMS handles messages on a subject for which a marshalling specification is present in the MIB, AMS automatically (un)marshalls the message content from a format suitable for the application use to a network neutral format and vice-versa.

**Communication model**

An AMS request may target a subset of all the nodes of an application. This subset is called the domain of the request. This subset is defined by the following filters:
• If the service request is one for which continuum ID is not specified, then only
nodes that are members of the local continuum are members of the domain.
Otherwise, if the service request’s continuum ID parameter does not indicate
“all continua”, then only nodes that are members of the continuum identified by
the service request’s continuum ID parameter are members of the domain.
• Only nodes that are members of the zone identified by the service request’s
zone ID parameter are members of the domain.
• If the service request’s role ID parameter does not indicate “all roles”\(^2\), then
only nodes performing the role identified by that role ID are members of the
domain.

To **send** a message is to cause it to be copied to the memory of a specified node.

To **publish** a message on a specified subject is to cause it to be sent to one or more
implicitly specified nodes, namely, all those that have requested copies of all
messages on the specified subject.

To **announce** a message is to cause it to be sent to one or more implicitly specified
nodes, namely, all those nodes that are members of a specified zone (possibly the
root zone) and that perform a specified role in the application (possibly “any role”).

A **subscription** is a statement requesting that one copy of every message published
on some specified subject by any node in the subscription’s domain be sent to the
subscribing node; the domain of a subscription is the domain of the AMS service
request that established the subscription.

An **invitation** is a statement of the manner in which messages on some specified
subject may be sent to the inviting node by nodes in the domain of the invitation; the
invitation’s domain is the domain of the AMS service request that established the
invitation.

Messages are exchanged directly between nodes rather than through any
intermediate dispatching server. They are automatically conveyed using the best
underlying transport layer connecting both nodes.

**Discovery**

AMS defines three types of communicating entities: nodes, registrars and
configuration servers:

• A registrar is a communicating entity that records informations about all nodes
of a single zone of a message space. It replies to queries for this information,
and it updates this information as changes are published.

• A configuration server is a communicating entity that records information about
the message space (i.e. the location of all registrars). It replies to queries for
this information, and it updates this information as changes are published.

Every zone is managed by a single registrar, which is responsible for monitoring the
health of all registered nodes in the zone. It is responsible for propagating node
registrations and terminations, subscription and invitation assertions and
cancellations. When it receives such a message from one node of its zone the
registrar immediately forwards this message to all other registered nodes in the zone,
and then to the registrars of other zones in the message space. Similarly, when it

\(^2\) The role name that is the character string of length zero indicates “all roles” or “any role”, whichever is less
restrictive in the context.
receives such a message from the registrar of an other zone, it forwards it to all the
nodes in its own zone.

Each registrar registers itself to the configuration server$^3$ for the continuum which
contains the zone. All registrars and nodes of the same message space must register
through the same configuration server.

When a new node joins the message space it has to register itself in a zone. As a
registrar can migrate over time, the node starts contacting the configuration server
which replies with the location of the corresponding registrar. The registrar allocates a
unique node number to the node, and then broadcasts the new node's information$^4$ to
all other nodes of the message space. All these nodes then announce themselves to
the new node.

It's important to maintain up-to-date knowledge about the configuration of the
message space, so each node automatically notifies its registrar when it stops. To
detect abnormal termination of nodes a heartbeat mechanism is used. In the same
way, each node monitors the registrar health and gets anew the zone configuration if
it detects a registrar crash. Each registrar is itself monitored by the configuration
server, which automatically takes action when it detects a registrar crash to get it
restarted.

The configuration server is monitored by all registrars. If it is missing, each registrar
tries to re-establish the connection using all known possible locations. When the
configuration server is restarted all registrars eventually find it and re-announce
themselves to it.

**Remote AMS message exchange**

It is common to group together in a single continuum all entities running in a similar
network. However, AMS messages can be easily exchanged between nodes in
different continua; this is the purpose of Remote AMS (RAMS) procedures.

RAMS procedures are executed by special nodes called RAMS gateways. All RAMS
gateways get organized in a RAMS grid that enables AMS messages to be forwarded
between message spaces. Each RAMS gateway has interfaces to two networks: the
AMS message space it serves and the grid of RAMS gateways.

**Security**

AMS can be configured to restrain the service access to authorized application
modules. The AMS MIB contains information$^5$ used to authenticate registrar and
nodes, and to assure the authenticity, confidentiality and integrity of the meta-
information traffic.

In addition, AMS MIB contains information allowing protection at subject granularity:

- A list of role names of all nodes authorized to send (resp. receive) messages
  on a given subject.
- Parameters allowing encryption of messages on this subject.

This information may be used to secure messages on the selected subject.

$^3$ A list of all possible locations for a configuration server is stored in the MIB.
$^4$ Node number, role name and location.
$^5$ Public Key.
3. JMS and Joram overviews

Message-Oriented Middleware (MOM) is increasingly being seen as a key to improve the enterprise productivity and to facilitate the open services market. Today enterprises are faced with the challenges of time-to-market, data distribution, application integration and business flexibility in the context of loosely-coupled distributed systems encountered in multi-organization environments over the Internet:

- Data integration, E-business, EAI and B2B solutions,
- Mobile users and ubiquitous systems,
- Networked devices management solutions: energy, building/home automation, RFID, etc.

Up to now, the lack of standard has been a strong obstacle to a wide adoption of asynchronous communication systems as a technical basis for implementing interoperability. Over the past few years the JMS™ API (Java Messaging System) has partially filled in this gap and has become a de facto standard for building cooperating software components over Internet based loosely-coupled distributed environments.

JORAM (Java Open Reliable Asynchronous Messaging) is the leading open source implementation of the JMS™ messaging specification worldwide. JORAM is an open-source software component available from ObjectWeb (http://joram.objectweb.org). JORAM is fully compliant with JMS 1.1⁶ and is available on a large range of computer systems, from application servers to internet appliances. J2ME™ applications, running on lightweight devices (e.g. PDA, SmartPhone, or any java-based embedded system), are able to use JORAM messaging capabilities to interoperate with JMS-based applications over Internet.

JORAM greatly benefits from the new generation Message Oriented Middleware from ScalAgent Distributed Technologies, an agent-based truly distributed architecture. The underlying innovative architecture allows distributed applications to be connected on a large-scale basis through Internet, enables load balancing and guarantees high availability and flexibility.

JORAM is available under the LGPL license, which is a very flexible certified open-source license. JORAM can thus be used in any product and can be freely redistributed.

3.1. The JMS Specification

JMS (Java Messaging Service) is the specification of a messaging service for Java applications. More precisely JMS describes the API that allows Java programs to communicate through an asynchronous communication system – i.e. sending and receiving messages. A detailed description of the JMS API is beyond the scope of this report, we present here only the JMS elements that are required to the understanding of architecture issues described later on.

A JMS application consists of the following elements (see Figure 1):

- A JMS platform (also called JMS provider) that provides the JMS run-time environment and a set of control and administrative functions.

---

⁶ As a component of the JOnAS J2EE application server, JORAM has completed the JMS/J2EE 1.4 compliance tests.
• The **JMS clients** are application programs, written in Java, that produce and consume messages according to the messaging protocols defined in the JMS API.

• **JMS messages** are entities that allow information to be conveyed between JMS clients. Different types of messages are supported in JMS: structured text (e.g. an XML file), binary data, java objects, etc.

Two communication models (called *Messaging Domains* in JMS) are supported:

- **Point to Point model**, based on **Queues** (Figure 2). A producer client sends a message to a message Queue where it is stored temporarily (flow 1). A consumer client may then read the message from the Queue to operate on it (flows 2-3 and 4-5). A given message is read only by a single consumer client (i.e. “point to point” model). The message stays in the Queue until it is read or the message time-to-live expires. Message retrieval from the queue can be requested explicitly by the consumer client or can be delivered automatically (calling a pre-defined “listener” function to be executed at the consumer site). Message consumption is then acknowledged either by the system or by the client application.

- **Multipoint model**, based on the **Publish/Subscribe** paradigm (Figure 3). A producer client publishes a message related to a pre-defined **Topic** (Flow 2).
All clients that have previously subscribed to this topic (flows 1a and 1b) are notified of the corresponding message (flows 3a and 3b).

```
Figure 3 - Publish/Subscribe Communication model
```

The latest version of the JMS specification (i.e. JMS 1.1) unifies the handling of the two communication models at the client level through the introduction of the Destination concept to represent a queue or a topic. This simplifies the API (i.e. the queue and topic functions are syntactically merged) and allows an optimization of communication resources. This unification does not change the semantics of each of the communication models that should be taken into account when programming the JMS client.

Finally the JMS specification defines some QoS options, especially for subscriptions (transient or durable) and for the guarantee of message delivery through the message persistency property.

**What JMS does not say and does not do**

JMS defines a communication protocol between message producers and consumers, but it does not give any hint about the way this protocol should be implemented by a given JMS provider. Therefore implementation issues stay outside the scope of the specification and all JMS messaging platforms are proprietary implementations that only conform to the JMS API.

A JMS application that complies to the JMS API is supposed to be independent of a given JMS messaging platform. Making the application work simultaneously on different JMS platforms requires the development of a specific gateway between the two platforms. This function is available on some JMS products. JORAM also provides this feature.

The JMS specification is not complete. Some key functions are not described, such as for example the administration of a JMS platform (i.e. deployment, configuration, monitoring, etc.) and are thus subject to proprietary implementations. At the opposite, most of the available products (commercial or open source) provide additional functions (not addressed in the specification) such as: hierarchical topics, security and high availability features, etc. On both issues JORAM is not an exception to the general rule as explained in the rest of the document.

---

7 In the rest of the paper the term « destination » will be used to designate indifferently a queue or a topic.
3.2. Joram overview

From the above description JORAM is a JMS platform. As any other JMS platform
JORAM is structured in two parts: the JORAM server that manages the JMS
abstractations (e.g. queues and topics) and the JORAM client that is bound with the
JMS client application. As we will see later on in the description of the JORAM
architecture, the JORAM server can be implemented as a central service or as a set
of cooperating distributed services. It should be noted here that the ability to deploy a
JMS platform as a distributed system with variable QoS parameters is a key issue
when comparing various JMS platforms.

Communication between a JORAM client and a JORAM server is generally relying on
TCP/IP\(^8\). Communication between JORAM servers is achieved through various
protocols according to application requirements: TCP/IP, HTTP or SSL.

Usually, clients and servers run on different machines. However they also can be
hosted on the same machine and run in different processes or share the same
process. In this case communications are optimized.

The sections below detail the key aspects of the JORAM platform architecture that
grants it its unrivalled flexibility and scalability properties.

Design Choices

The main characteristic of JORAM is its distributed configurable architecture. Basically
JORAM has adopted a snowflake architecture - i.e. a JORAM platform is composed
of a set of JORAM servers interconnected by a message bus that offers various
communication protocols. Each server can manage a variable number of JMS clients.

---

\(^8\) In addition HTTP/SOAP may also be used for lightweight JMS clients developed in the J2ME environment and
SSL can be used for secure communications.
level concerns the positionning of communication objects (Queues and Topics). Note that these parameters have a deep impact on performance figures and scalability and availability issues. A third level of configuration consists in defining the QoS parameters (communication protocol, persistency, security, etc.). The choice of a given mechanism is always a tradeoff between the expected QoS level and the cost of the solution. The overall architecture of a JORAM platform is depicted in Figure 4.

**Logical Architecture**

This section describes the principles of a communication path between two JMS clients for both the Point to Point and Publish/Subscribe communication models. The presentation aims at pointing out the data and control flows involved in these exchanges. Distribution is not depicted in this functional description (i.e. in a given implementation the Destination and Proxy objects may run on different servers).

_connection, Session and Sender_ (respectively _Receiver_) are temporary JMS objects created by a JORAM client when a connection is established with the server9. On the server each JMS client is represented by a persistent _Proxy_ object that is created by the server whenever a new user is registered. A proxy object implements basically two functions:

- Communications management between the server and the client.
- Message delivery to (resp. message retrieval from) the destination.

**Point-To-Point model**

The communication between a producer client and a consumer client is achieved through the following steps (described in Figure 5 by the red arrows).

When a Send operation is executed on the producer client site, a JMS message is sent to the corresponding Proxy object (called Proxy-P) within the server (plain arrow 1).

The Proxy-P object encapsulates the JMS message into a MOM message to be transported by the MOM to the destination site (i.e. the node of the Queue object). The MOM message is saved on a local persistent store. An acknowledgment is returned to the JMS client (dotted arrow 2). As seen from the producer client the Send operation is over and the client may continue its execution asynchronously while the message is actually delivered.

The MOM message is delivered to the Queue object (arrow 3). This may require a communication between two servers if the Proxy-P and the Queue Objects are not handled by the same server. The proxy object implements a Store and Forward function, which plays a key role in communication reliability (this feature is addressed below). Finally the MOM message is saved on a local persistent store that implements the message queue.

---

9 The role and use of JMS objects is not detailed here. They are available in the JMS specification.
When a Receive operation is executed at the consumer client site a control message is sent to the corresponding proxy object (noted Proxy-C) in the server which forwards it to the Queue object. As in step 2, this may require a communication between two servers if the Proxy-C and Queue objects are not settled on the same server. The MOM message is retrieved from the Queue object and sent to the Proxy-C object which, in turn, extracts the JMS message from its envelope and returns the JMS message to the consumer client. This set of operations is described by the set of plain arrows 4. At this stage the Receive operation is over.

An acknowledgment is returned by the JORAM client to the Queue object in order to free the resources on the server that manages the Queue object (dotted arrows 5). The Ack message may be sent explicitly by the JMS client or generated by the JORAM client.

In step N° 2 the MOM message is stored before being delivered to the Queue object. This allows the delivery operation to be re-executed if it fails, for example because of the receiver server being not available (step 3). This feature, referred to as the Store and Forward function, guarantees the message delivery at the JORAM level. It should be noted that we have not described here the exchanges that occur at the MOM level.

**Publish/Subscribe Model**

Figure 6 below depicts the control and data flows for the Publish/Subscribe communication model.

Steps 1 and 2 are similar to those of the point to point communication model.

In step 3, the message built by the producer proxy object (Proxy-P) is delivered directly to the whole set of consumer proxy objects (Proxy-C) where they are stored. This is a key difference with the prior schema as the Topic object is not a final destination but merely a switch/router towards the set of consumer proxy objects.

The consumer operation is implemented by an exchange between the client and the Proxy-C object (plain arrows 4). The acknowledgment is then returned to the Proxy-C object (dotted arrow 5). By comparison with the preceding schema it can be noted that the consuming dialog is restricted to exchanges between the client and its proxy.
object. No dialog occurs with the Topic object itself. This usually leads to shorter interactions and better performance from the consumer point of view.

The next two following sections describe how this logical schema is implemented in a centralized architecture (i.e. a single JORAM server) in a first step, and in a distributed architecture in a further step.

**Centralized Architecture**

In this configuration, illustrated in Figure 7, all JMS clients are connected to a single JORAM server that manages all destination and proxy objects. The communication protocol is simplified as all communicating objects are located on the same server. Administration and operation are also greatly simplified.

---

10 In this figure, the data flow for the message production is described with plain arrows, while the data flow for the message consumption is described by dotted arrows. Acknowledgments are not represented.
At this stage, as stated above, we define a simple JORAM platform as composed of a single server hosting the following services:

- A connection manager to handle the connections with the clients through proxy objects. This service initializes the local administration module and may also create an administrator who is authenticated by his name and password.
- A proxy for each connection mode used by a client – here a TCP proxy listening on a predefined port.

The platform configuration is described in an XML file. Figure 8 gives an example of a minimal configuration made of a single server. This server, numbered 0, is named “S0” and is hosted on the "localhost" machine. The server is hosting the connection manager (ConnectionManager) and a TCP proxy listening on port 16010. In addition an administrator, identified as root – root, is assigned to the connection manager.

```xml
<?xml version="1.0"?>
<config>
  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.JORAM.mom.proxies.ConnectionManager" args="root root"/>
    <service class="org.objectweb.JORAM.mom.proxies.tcp.TcpProxyService" args="16010"/>
  </server>
</config>
```

This centralized solution has two major drawbacks: lack of availability and lack of scalability. The server is a single-point of failure that prevents the whole system to run if a failure occurs on the server side. Computing and storage resources are all concentrated on a single machine. This may result in serious performance bottlenecks when the load increases (e.g. number of clients, number and size of messages, etc.).

**Distributed Architecture**

In a distributed architecture approach several instances of the JORAM server cooperate, each of them being in charge of a set of JMS clients. This architecture is
illustrated in Figure 9 for a scenario composed of three physical servers (to simplify the figure, only one JMS client has been represented at each location)\textsuperscript{11}.

The JMS message produced by the Client-1 JMS client towards Q3 message queue is sent to the Px1 proxy object on server S1. The JMS message is embedded within a MOM message. The Store and Forward function implemented in the proxy stores the MOM message locally before forwarding it to the S3 server where queue Q3 is located. When stored in Q3 the message can then be retrieved by a JMS client connected to server S3. In the same way, a message produced by the Client-2 JMS client to the Q1 message queue is delivered to S1 through the Px2 proxy object in server S2.

The physical communication between the servers is achieved by the underlying distributed MOM that guarantees the delivery of messages at the MOM level despite transient network and server failures.

Figure 9 - Distributed architecture

A distributed JORAM platform involves several JORAM servers. To interoperate, these servers must belong to a given administration domain and the following JORAM services have to be available on the various servers:

- A connection manager to manage the connections with the clients connected to that server using a set of user proxies. This service initializes the local administration module and possibly creates an administrator authenticated by a name and a password. At least one of the servers within a distributed platform must support the creation of an administrator.
- A user proxy specialized for each type of connection used by a client that interacts with the server – in our case a TCP proxy listening on a pre-defined port. This service is required on all servers that manage TCP connections with their clients.

In order to deploy the architecture described above we have to describe each server with its own services. We have also to define a communication domain (D1) and, for each of the servers, the access point to this domain (Network).

\textsuperscript{11} This particular configuration is provided as an example. The positioning of Queues and Topics is achieved by the system administrator.
JORAM: a Highly Configurable JMS Platform

Building a JMS platform tailored for a given application context is a difficult task as the resulting architecture is based on complex tradeoffs between numerous evaluation criteria such as: performance, availability, scalability, flexibility and evolution, security, development and operation costs, etc.

Therefore the role of the architect is crucial. Based on a set of application requirements he has to take into account the evaluation criteria mentioned above and to give them a specific weight in order to design the architecture that better fits the requirements. This work is achievable if and only if the messaging system provides the configuration capabilities that enable such a design process. JORAM answers this need through a combination of configuration and tuning capabilities available at various levels:

- Overall organization of JMS servers and clients. As depicted in Figure 4 a JORAM platform is structured according to a “snowflake” architecture. This approach gives to the application architect the freedom to settle servers where they are needed – e.g. to locally serve a set of geographically distributed JMS clients (edge computing), or to answer security constraints, or any other relevant criteria.

- Positionning of JMS objects. The positionning of JMS communication objects (i.e. queues and topics) is a key issue as it has a strong impact on the overall performance as well as on client availability.

- Servers sizing (computing power, storage, communication facilities). The JORAM Web Site provides some benchmarking figures that can help application designers to anticipate the sizing of the various JORAM servers to support the load generated by their local clients.
• System extension to support scalability. This parameter refers to the ability for a JORAM platform to grow dynamically to answer evolving application requirements (e.g. new messaging server, increasing load, etc.). Management facilities are provided to remotely add and remove JORAM servers dynamically through the Administration API of JORAM.

• Communication protocols. Various communication protocols are available between clients and their server as well as between two servers: TCP/IP, HTTP, SSL, SOAP, etc.

• QoS parameters (e.g. persistency, security). Messages transfer reliability (i.e. guarantee of message delivery) and security (message confidentiality) have a cost and the application designer can select the right option for his application.

• Level of availability through clustering and replication. JORAM answers the needs of mission critical applications with the ability to design clustered destinations and to deploy high available JORAM servers. Theses features are briefly introduced in the next section.

Today very few platforms provide a level of flexibility comparable to that of JORAM. This is obviously a major advantage compared to concurrent products.

4. Comparative study

This chapter describes the main similarities and differences between AMS and JMS. As JMS is only an interface specification and AMS covers a bit more of some implementation aspects, we will extend this study with a comparison of AMS and Joram (i.e. a JMS implementation).

To begin with, even if AMS and JMS share some similar goals, i.e. asynchronous communication between loosely coupled components, their targets seem really different: J2EE application over Internet for JMS versus embedded autonomous system over proprietary networks for AMS.

JMS offers some capabilities not offered by AMS. Distinctive JMS capabilities include for example point-to-point delivery to exactly one of many consumers, message priority and time-to-live, and enterprise specific features such as full transactional support, and application level acknowledgements.

AMS provides a decentralized peer-to-peer architecture with spontaneous discovery, which can be more robust and efficient compared to centralized server based architecture commonly used for JMS.

4.1. Architecture

While AMS defines a complete architecture framework for the applications, JMS specification focuses on a client/server model without any constraint about the applications or the provider architecture:

• The AMS architecture is peer-to-peer: each distributed component potentially interacts directly with every other component and maintains a network reference to each component.

• The JMS architecture is peer-to-peer in the sense that, at least potentially, every client can communicate with every other. However with JMS, distributed components interact with the JMS broker via destinations (topics and queues),
which then disseminates messages to the registered clients; thus, distributed components do not have to know and manage each other's network locations.

A JMS provider can use (behind the scenes) any of several network topologies, for example, bus, ring, hierarchical, fully connected (peer-to-peer), star (hub-and-spoke), or others, including hybrids.

Centralized/Distributed vs. Peer to Peer messaging

In a Peer-To-Peer system every messaging component maintains a connection to every other component. It allows to implement dedicated and more efficient communications, however implementing the reliability and security functions requires more complex algorithms and needs more resources in each component. Each component needs to know the overall architecture, it implies numerous configuration messages: (un)register, (un)subscription, invitation, etc.. As components are added to the system, the number of connections rises exponentially. This makes asynchronous message delivery and scalability difficult to achieve.

Centralized management is also problematic; the preferred approach for enterprise messaging is a distributed messaging system. In this approach each messaging component maintains a connection to one message server. All message servers cooperate to provide the messaging service. The messaging service ensures routing and delivery of messages between components, and is responsible for reliable delivery and security. Components interact with the message service through a well-defined programming interface. As components are added to the system, the number of connections rises only linearly, making it easier to scale the system by scaling the message service.

Conclusion

AMS offers a Peer-To-Peer architecture, it needs a costly and complex distributed mechanism to maintain up-to-date knowledge about the configuration. At the opposite it allows direct component to component communication. In order to reduce this complexity and increase the scalability AMS introduces the segmentation of the message space; it then needs RAMS gateway.

Joram offers a distributed configurable architecture; typically a Joram platform is composed of a set of Joram's servers interconnected by a message bus. Each server can manage a variable number of clients. The number and location of servers is configurable; at the extreme degree a server can be located at each client location and Joram looks like a peer-to-peer system.

4.2. Message Format

JMS provides five predefined message types to specify different types of message payload. The JMSType header field contains a message type identifier supplied by a client when a message is sent. JMS does not define a standard message definition repository.

AMS does not provide any predefined message type. However, marshalling information can be provided for message content in the AMS MIB.

JMS allows specifying properties for each message: priority, time-to-live, or application's properties.
Conclusion

The JMS message structure inherits from a long MOM experience, it allows powerful processing either from the provider or the application. The AMS message is well fitted for embedded systems with restrained resources. In JMS heterogeneity handling is based on the Java universality, while the marshalling informations contained in the MIB may allow AMS to work with different languages assuming that the required tools are provided with the AMS implementation.

4.3. Communication model

Publish/Subscribe communication model

AMS and JMS both provide the Publish/Subscribe communication model. The P/S model uses asynchronous message passing between concurrent subsystems. It connects anonymous data producers with data consumers. This communication model enables a robust service based application architecture that decouples each component from another; it provides location transparency, and the flexibility to dynamically add or remove component.

AMS allows message routing either through subject (or "topic") selection, or using node's properties: role or including context. JMS message routing is done through a topic in a static hierarchy.

A JMS topic acts as logical messages channel managing the delivery of messages sent to it. A consumer is attached to a specific destination from which it receives message, a producer can specify the destination at the time of sending the message.

In AMS messages are dynamically routed from the sender to the receivers by the middleware using the knowledge of existing subscriptions.

Point-To-Point communication model

In the JMS Point-To-Point messaging domain, a destination (Queue) may have multiple consumers and producers. A message is processed by exactly one of the attached consumers. Thus, a message is delivered point-to-point, from the producer to one of the many available consumers. The policy for selecting a consumer is left up-to the middleware provider (first requesting). The point-to-point delivery mechanism in the JMS PTP messaging domain makes it very easy to distribute processing load across multiple identical consumers, thus offering a simple way to load balancing.

Since PTP behaves as if the messages are put in a single logical queue and handed over to one of the available consumers, this messaging domain will generally be less scalable that the P/S domain.

AMS does not support this point-to-point delivery mechanism. All the matching consumers associated with a subject will receive updates. However some AMS communication methods may be used to partially achieve point-to-point delivery.
**Conclusion**

JMS and AMS each offer an asynchronous communication mechanism between messages producers and consumers; however AMS does not provide a Point-to-Point delivery model essential for a large class of applications.

### 4.4. Acknowledgment and transaction

JMS defines different acknowledge modes, implicit or explicit, that allows multiple quality of service for message delivery: transactional, once and only once, best-effort, etc. JMS attempts to redeliver a message until the receiver acknowledges it. In case of failure during the message handling the message is kept by the provider to be redelivered.

JMS supports transacted sessions, allowing a group of operations\(^{12}\) to be treated as a single unit of work. JMS also allows full transactional support enabling transactional interaction with the client code. This may be essential when the exactly once semantics is to be ensured up to the application level. This kind of interaction is quite usual in the context of backoffice IS applications.

AMS only supports the "best effort" delivery mode.

**Conclusion**

The AMS best-effort mode enables messages to be transferred with minimal latency; it seems well suited to the needs of some applications but this limited model will be a weakness for some others. Transactional support should be provided by external components by dint of much development efforts.

### 4.5. Persistency and durability

Persistency refers to the ability of specifying message delivery so that it survives middleware failures. With persistent delivery, an application is assured that when a send or a write operation returns, the middleware will not lose the data even if it crashes.

Durability refers to the ability of a consumer to receive data sent to it even when it is not active. In JMS when a durable consumer restarts, for example after a crash, it receives any messages sent to it (while it was inactive).

JMS provides independent mechanisms for controlling both persistency and durability. In JMS durability is specified by creating a durable consumer, either by creating a QueueReceiver, or by calling Session.createDurableSubscriber().

Persistency is specified by the delivery mode used to send message. For PERSISTENT delivery mode, a message is stored on permanent storage before the send method returns to the caller; the message is delivered once and only once (redelivered messages are marked). For NON_PERSISTENT delivery mode, a message is not stored before the send operation returns to the caller; the message is delivered at-most-once, allowing for the possibility of losing a message after the send operation has completed, but before it can be delivered because of a middleware failure.

\(^{12}\) Send or receive methods.
Due to its peer-to-peer architecture, AMS does not allow to control neither persistency nor durability.

**Conclusion**

Persistency and durability are two main properties of Message Oriented Middleware; they simplify the handling of failures quite usual in large and loosely coupled distributed systems. Their missing in AMS will probably require many applications to build similar functions on top of AMS.

### 4.6. Discovery

JMS discovery is administered and logically centralized. JMS discovery requires that the producers and consumers be able to find and bind to the destinations (and not to each other). There are two mechanisms for JMS destination discovery.

Normally, destinations are dynamically discovered using the JNDI APIs, which bind logical destination names to destination objects. The discovery issue is then delegated to the JNDI implementation. In order to avoid making JNDI a critical resource, it should be implemented as a distributed and highly available service. Destinations may also be discovered via attributes of received messages. Destinations may also be statically defined. They must then be created and configured before a client can use them. Determining what static destinations to use is a critical aspect of a distributed system design, and must be considered carefully prior to deploying a system based on JMS. Evolving the system configuration for new requirements also requires careful planning and administration. Destinations take up physical resources, so destinations no longer needed in the distributed system must be purged, and new ones added as needed over the lifetime of a distributed system based on JMS.

AMS discovery is spontaneous and decentralized. However, the role of the AMS MIB is not really clear and its seemingly static nature looks like a brake upon application evolution.

**Conclusion**

The AMS’s discovery is implied by the peer-to-peer architecture, it needs many message exchanges with an extra cost for the overall solution (see the discussion about architecture 4.1).

### 4.7. Security

The JMS specification does not define a security model, there is just an authentification step during connection.

Joram provides a two levels security model, allowing SSL secured connections between client and servers to ensure authentification and data integrity, and defining a protection model to control the use of sending and receiving operation. It is quite comparable with the AMS security model.
5. JMS over AMS, AMS over Joram

5.1. JMS over AMS

As described in the previous chapter there are several differences between AMS and JMS. However we will study the ability to implement the JMS API on top of the AMS. Below we describe the main difficulties to solve.

JMS Message model

The AMS messages are lightweight entities when JMS messages consist of a header and a payload: the header contains fields used for message routing and identification; the payload contains the application data being sent.

Even, if we can wrap JMS message into AMS messages, it seems difficult to offer some of the JMS functions such as Message selection or priority.

Message selection

As there is no intermediate entity, queue or topic, in AMS the message filtering must be done by the client on message receipt. As described in the JMS specification (paragraph 3.8), this is not really satisfactory:

| JMS provides a facility that allows clients to delegate message selection to their JMS provider. This simplifies the work of the client and allows JMS providers to eliminate the time and bandwidth they would otherwise waste sending messages to clients that don’t need them. |

JMS Publish/Subscribe model

Despite numerous differences the JMS Publish/Subscribe model can be adapted to the AMS communication model. The topic JMS concept can be easily merged with the subject AMS concept.

However, as described in the JMS specification (paragraph 6.3), a JMS subscriber can be made durable:

| A durable subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the prior subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription’s messages until they are received by the subscription or until they expire. All JMS providers must be able to run JMS applications that dynamically create and delete durable subscriptions. |

It seems difficult to implement this function in AMS, a solution could be to use an external node to handle the persistancy of such subscriptions.
JMS Point-To-Point model

AMS does not provide a Point-To-Point communication model. Moreover, the JMS Point-To-Point model is based on the concept of Message Queues that act as a mailbox and store messages. Due to its Peer-To-Peer architecture there is no entity in AMS to play this role, apart implementing the function in a dedicated client.

Conclusion

Resulting from the paragraphs above, it seems difficult to implement a fully compliant JMS API on top of AMS. AMS is designed as a message bus and well suited for the Publish/Subscribe paradigm; the functions required for message queuing do not fit easily with it.

However a JMS like interface restricted to the Publish/Subscribe domain could be built on top of AMS. This interface should offer a known API to Java clients.

5.2. AMS over Joram

At the opposite of JMS, the AMS specification is not limited to the client API but it defines the overall behavior of the system. As these interactions are defined by taking into account the peer-to-peer architecture of AMS, it is a little bit difficult to propose an alternate implementation. As for subscriptions, whereas JMS specifies that a subscriber must receive all messages sent on the topic, AMS specifies that all nodes have to be informed of the new subscription in order to allow the broadcast of messages to the new subscriber. Due to the Joram's architecture a realization of AMS over Joram cannot respect this implementation level requirement, but will focus on offering the functional interface.

This study can be separated into two lines; the first concerns the functional realization of the communication functions of AMS on top of the JMS/Joram concepts, the second implies the realization and optimization of the communication channels.

Three implementation tracks may be followed. In solution 1 we consider implementing AMS on top of the mere JMS API. Solution 2 is a variant of solution 1, taking into account the JORAM implementation of JMS. Solution 3 makes a direct use of JORAM's internals.

Solution 1

In this first solution we respect as scrupulously as possible the operating mode specified by AMS without benefiting from the assets of JMS/Joram. Each context is represented by a topic on which configuration messages (registering, subscription, and invitation) are emitted.

These topics are organized according to a tree structure duplicating the architecture of the message space. Each node listens to the topic corresponding to its context and updates its configuration view of the system. It receives the configuration messages from all the nodes in the system and can thus determine the subset of the recipient nodes for each message it will send (publish, send, announce).
In addition, each node is represented by a queue which collects messages sent to the node. The sending node can then transmit the message in PTP to the queue of each node in the recipient subset.

This solution preserves the internal functioning described in the AMS specification, by using JMS as medium of communication. It inherits the main disadvantages of a peer-to-peer architecture, deteriorated by the cost of the interaction with the provider (instead of a direct communication):

- Many configuration messages, each node having to preserve a precise image and up-to-date architecture of the application.
- Broadcast carried out directly by the transmitter and possible emission of many messages on weak links.

**Solution 2**

The first solution strictly respects the AMS specification, in terms of both interfaces and implementation (description of internal messages). We try here to take advantage of the Joram/JMS behavior to implement a system respecting the AMS interfaces and execution properties, while improving its overall efficiency and scalability.

As in the previous solution, each context is represented by a topic, and every node is represented by a queue. Each subject is associated with a topic. These destinations are used only for the routing of the functional messages. The configuration messages are transmitted towards one (or more) specific destination and handled by a broker implementing the function of discovery fulfilled by the registrar and the configuration service of AMS. This broker receives the monitoring messages from nodes and applies the corresponding actions to the repository: creation (resp. deletion) of a queue during node registering (resp. unregistering), etc. The actions of broadcast (publish, send, announce) and reception are directly translated into JMS actions.

![Message space](image)

*Figure 11 – AMS zone and Joram hierarchical topics*

Messages are routed through a joint use of hierarchy of destinations and message selection. For example:

- In the example above (Figure 11), there are three nodes in different zone of a message space. When publishing a message (*Publish.request*), every node can simply send the message to the topic corresponding to its zone. It fixes some properties in the message to allow message selection by subscribers: subject, role ID, etc.
Let us assume that each of these nodes has subscribed to a given subject "S" (*Assert_subscription.request*), X from all subscribers, Y and Z from the subscriber of their own zone. In order to receive the waiting messages, X must subscribe to topic "root", Y and Z to the topic corresponding to their zones.

This solution leads to decreasing the diffusion of the configuration messages. We can handle the private messages (*Send.request*, *Announce.request*) in a similar way, using the sender and receiver properties to restrain the diffusion of messages.

As an alternate solution, the realization of a specialized destination whose semantics takes into account the concepts of zones and roles would improve the implementation. In the same way the configuration broker could be an internal component of Joram making it possible to reduce the costs of communication and to facilitate its administration.

**Solution3**

Another more complex solution could be to build a client AMS and its associated proxy on top of the ScalAgent MOM (as it is done for JMS with Joram). This approach would better exploit the AMS characteristics (discovery, routing, etc.). However it involves two main drawbacks, the cost of realizing it, and a strong dependency on JORAM.

**Communications aspects**

Numerous aspects in the AMS specification aim at minimizing communication cost: "best-effort" delivery mode that does not require acknowledgments, reduced message overhead, direct peer-to-peer message transfer between sender and receiver. We will consider the Joram architectures to similarly maximize performance and scalability.

1. First of all we can use the distributed architecture of Joram to place servers at strategic location and to decrease the data flows on the communication links. Joram offers many possibilities of interconnection over IP (others can be realized if needed), direct communication can be obtained by means of embedded servers (colocated). It allows the use of dedicated protocols according to the domain of use.

2. The configuration of Joram allows to optimize the interactions between client and server, we can so obtain a best-effort mode in the detriment of the conformance with JMS. Finally, the publish/subscribe mode of Joram is optimized to remove additional cost due to the persistency when using non-durable subscriptions.

3. Additional cost due to the rich structure of JMS messages will be partially removed thanks to an ad'hoc serialization protocol. It could be further decreased by the development of an AMS specific proxy (as described above) and a wrapping between JMS and AMS messages. This overcost would thus be limited to JMS internals communication and would impact neither the client nor communication between client and provider.

**Conclusion**

The realization of AMS on top of Joram can be done in several ways depending of the objectives. The second solution seems to offer the best compromise between the realization cost, the overall performance and the respect of JMS and AMS semantics.
6. Conclusion.

The design of AMS is strongly dependent on the constraints of the applications concerned and their environment. AMS answers mainly to the concept of publish/subscribe, messages are routed depending on the architecture of the application (composition of the messages spaces). Compared with JMS, AMS has several features that can potentially minimize communication costs; these include reduced message overhead and direct message transfer. The AMS API does not provide the same transparency level than JMS, especially it needs numerous (optionally) indication methods to facilitate message space monitoring.

JMS is a more general API designed to give Java applications access to a large variety of Message Oriented Middlewares (MOM). A JMS Message is a rather heavy structure which looks not well suited to the embedded and space world where the resources are limited. JMS allows the interaction either with “message queuing” systems or publish/subscribe ones. It includes many properties of quality of service making it possible to adapt to a broad range of applications: time-to-live, persistency, transactions, etc. The routing of the messages is related to the logical concept of “destination” and independent of the localization of the producers and consumers.

Joram is an open-source implementation of JMS based on an adaptable MOM: it is available on a wide range of computer systems and networks, from application servers to internet appliances. An AMS implementation on top of Joram should benefit from both the JMS API and architecture (well recognized) and the Joram’s capacity to fit to complicated architecture.
1. Introduction.  

2. AMS overview  

2.1. AMS model  

2.2. AMS architecture  

<table>
<thead>
<tr>
<th>Message format</th>
<th>Communication model</th>
<th>Discovery</th>
<th>Remote AMS message exchange</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

3. JMS and Joram overviews  

3.1. The JMS Specification  

<table>
<thead>
<tr>
<th>What JMS does not say and does not do</th>
<th>8</th>
</tr>
</thead>
</table>

3.2. Joram overview  

<table>
<thead>
<tr>
<th>Design Choices</th>
<th>Logical Architecture</th>
<th>Centralized Architecture</th>
<th>Distributed Architecture</th>
<th>JORAM: a Highly Configurable JMS Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

4. Comparative study  

4.1. Architecture  

<table>
<thead>
<tr>
<th>Centralized/Distributed vs. Peer to Peer messaging</th>
<th>17</th>
</tr>
</thead>
</table>

| Conclusion | 17 |

4.2. Message Format  

| Conclusion | 18 |

4.3. Communication model  

<table>
<thead>
<tr>
<th>Publish/Subscribe communication model</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-To-Point communication model</td>
<td>18</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.4. Acknowledgment and transaction</td>
<td>19</td>
</tr>
<tr>
<td>Conclusion</td>
<td>19</td>
</tr>
<tr>
<td>4.5. Persistency and durability</td>
<td>19</td>
</tr>
<tr>
<td>Conclusion</td>
<td>20</td>
</tr>
<tr>
<td>4.6. Discovery</td>
<td>20</td>
</tr>
<tr>
<td>Conclusion</td>
<td>20</td>
</tr>
<tr>
<td>4.7. Security</td>
<td>20</td>
</tr>
<tr>
<td>5. JMS over AMS, AMS over Joram</td>
<td>21</td>
</tr>
<tr>
<td>5.1. JMS over AMS</td>
<td>21</td>
</tr>
<tr>
<td>JMS Message model</td>
<td>21</td>
</tr>
<tr>
<td>JMS Publish/Subscribe model</td>
<td>21</td>
</tr>
<tr>
<td>JMS Point-To-Point model</td>
<td>22</td>
</tr>
<tr>
<td>Conclusion</td>
<td>22</td>
</tr>
<tr>
<td>5.2. AMS over Joram</td>
<td>22</td>
</tr>
<tr>
<td>Solution 1</td>
<td>22</td>
</tr>
<tr>
<td>Solution 2</td>
<td>23</td>
</tr>
<tr>
<td>Solution 3</td>
<td>24</td>
</tr>
<tr>
<td>Communications aspects</td>
<td>24</td>
</tr>
<tr>
<td>Conclusion</td>
<td>24</td>
</tr>
<tr>
<td>6. Conclusion</td>
<td>25</td>
</tr>
</tbody>
</table>