# JBossESB requirements and architecture

Version: 0.3

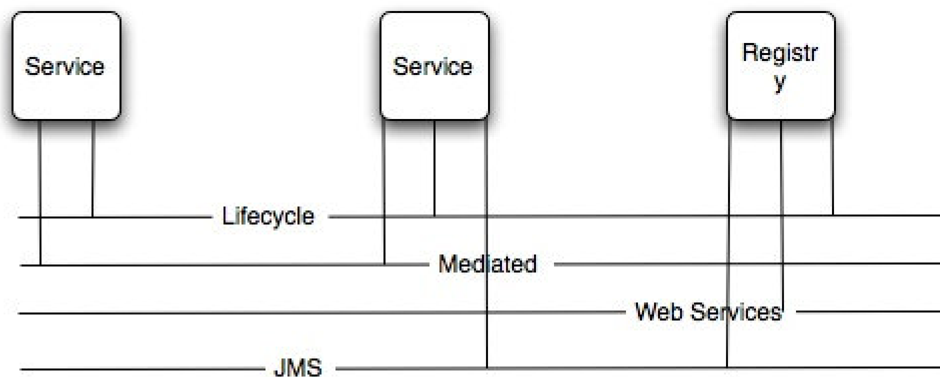
Date: 5/2/2006

## 1. Introduction

This document represents an outline of the overall requirements for JBossESB and the architectural approach we shall take. It should be noted from the start that the main aim of JBossESB is to provide an SOA infrastructure for deployment, runtime and management. SOA principles will be used throughout the architecture, with the traditional concept of an ESB being one possible narrowing of the capabilities it provides.

## 1.1 Overall goals

Enterprises will always be heterogeneous; legacy systems are here to stay and will continue to grow. The ESB is the best solution to bridge these technologies by leveraging SOA. The best way in which is can accomplish this is to abstract all of the components – JBossESB will not mandate any implementations because it then becomes part of the legacy problem. SOA principles will be used *internally* to JBossESB as well as *externally*: *everything* will be a logical service and at the architectural level interacted with via *messages*.




For example, the diagram above shows the important aspect of the JBossESB architecture: everything is a service, including the bus, and all services are interacted with via messages. As such, even though at an implementation level services may reside within containers responsible for lifecycle management (e.g., init/destroy, start/stop, suspend/resume), it will be considered architecturally as though those services were plugged directly into a lifecycle bus and receiving appropriate messages. This figure also illustrates how services can be plugged into multiple buses concurrently.

If the JBossESB implementation is suitably architected, it should be possible to switch in different core services, such as micro-containers to provide the various capabilities. From an architectural perspective, it shouldn't really matter what implementations comprise the runtime framework. This is the approach we will take in JBossESB. The default, out-of-the-box configuration will be pure JBoss, but this can be customizable. Furthermore, heterogeneity of ESB implementations will be the normal situation. Therefore, the "bus" has got to be able to interoperate with other "buses": think of what we want to achieve as a global, virtual ESB.

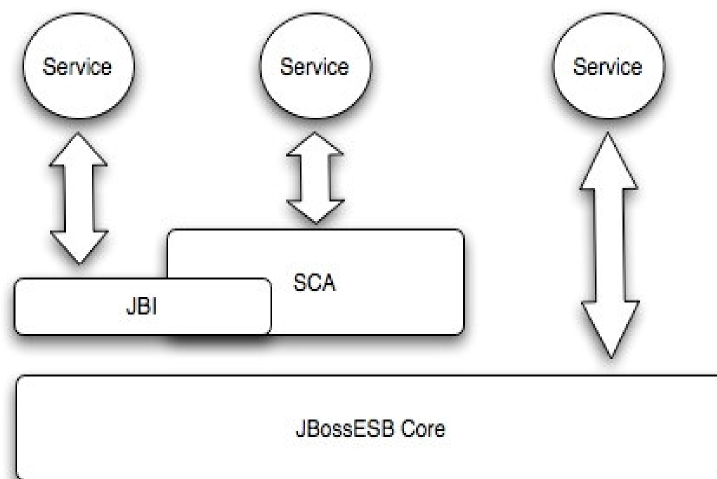
JBossESB will provide a basic core ESB framework that allows everything else to be configurable. Plug-in modules/components provide the real meat of the product, but all of them will be swappable by end users. Interdependencies will not be allowed in a hard-coded manner, but should be taken care of within the framework in an abstract (and configurable) manner. At the limit, it should be possible for someone to replace every single component of the ESB.

The concept of a single bus (product) that rules the enterprise is wrong and counter to both Web Services and SOA. In fact this was one of the biggest problems with the old style of EAI, which resulted in a lot of legacy systems as vendors moved from one implementation (product) to another, unable to leverage their current investments. We need to acknowledge that there will be other ESBs with which we will be interacting, potentially at all levels: federating of the ESB infrastructure should

be assumed from the start. A single bus does not scale and may become a bottleneck; as in hardware, multiple buses should be defined for different purposes and priorities.

The real differentiator for ESB/JBI vendors will be at the higher level, where the JBI specification does not go. It is through added-value features that vendors will make money. The real value of a JBI/ESB implementation will be in its ability to act as the unifying SOA deployment framework.

Standards are important. However, history has shown that standards/specifications come and go, whereas *requirements* persist. So, for example, in the case of transactions, we've had XA, OTS, TIP, BTP, WS-CAF, WS-TX, JTA, but at their core they are doing the same thing. Messaging is another good example: CORBA messaging, JMS, WS-RX, etc. Hence the reason that our first target will be to produce a core ESB framework that matches requirements and ignores specifications in this area: we want something that will last and won't require re-implementation several years from now. Therefore, ESB and SOA standards such as JBI and SCA will be layered on top of this. For example, SCA layered on JBI or SCA layered on native JBossESB interfaces.




## 2. The ESB requirements overview

The ESB is seen as the next generation of EAI – better and without the vendor-lockin characteristics of old. As such, many of the capabilities of a good ESB mirror those of existing EAI offerings. Traditional EAI stacks consist of: Business Process Monitoring, Integrated Development Environment, Human Workflow User Interface, Business Process Management, Connectors, Transaction Manager, Security, Application Container, Messaging Service, Metadata Repository, Naming and Directory Service, Distributed Computing Architecture.

As with EAI systems, ESB is *not* about business logic – that is left to higher levels. It is about infrastructure logic. Although there are many different definitions of what constitutes an ESB, what everyone agrees on now is that an ESB is part of an SOA infrastructure. However, SOA is not simply a technology or a product: it's a style of design, with many aspects (such as architectural, methodological and organisational) unrelated to the actual technology. But obviously at some point it becomes necessary to map the abstract SOA to a concrete implementation and that's where the ESB comes in to play.

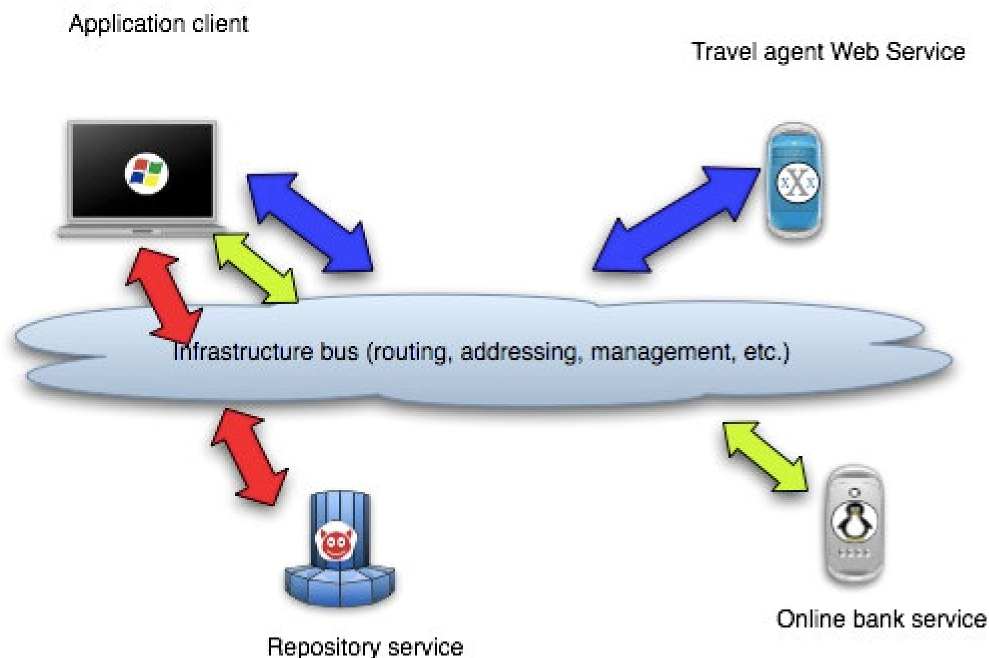
By considering ESB in terms of an SOA infrastructure, then we have the flexibility to abstract away from given implementation choices, such as JMS, SOAP etc. Then we define the capabilities that we want from our SOA infrastructure, which become the capabilities for the ESB. However, because of their heritage, ESBs typically come with a few assumptions that are not inherent to SOA:

- Java specific.
- Run-time message mediator.
- Message translation.
- Security model translation.

Loose coupling *does not* require a mediator to route messages, although that is dominant ESB architecture. This is also a requirement within the JBI specification. The ESB model should not restrict the SOA model, but should be seen as a concrete representation of SOA. As a result, if there is a conflict between the way SOA would approach something and the way in which is may be done in a traditional ESB, the SOA approach will win.

Therefore, in JBossESB, mediation will be a deployment choice and not a mandatory requirement. Obviously for compliance with certain specifications it may be configured by default, but if developers don't need that compliance point, they should be able to remove it (generally or on a per service basis).

The abstract view of the ESB/SOA infrastructure is shown below:




At its core, a good SOA should have a good MOM infrastructure, and JMS is a fairly good example of a standards-compliant MOM. But it obviously will not be the only implementation supported. Other capabilities that an ESB provides include:

- Process orchestration, typically via WS-BPEL.
- Protocol translation.
- Adapters.
- Change management (hot deployment, versioning, lifecycle management).
- Quality of service (transactions, failover).
- Qualify of protection (message encryption, security).
- Management.

Access control lists (ACLs) are important and complimentary to security protocols, such as WS-Security/WS-Trust, and often overlooked by existing implementations. JBossESB will support ACLs are part of the security capabilities.

Many of these capabilities can be obtained by plugging in other services or layering existing functionality on the ESB. We should see the ESB as the fabric for building, deploying and managing event-driven SOA applications and systems.

As mentioned earlier, there are many different ways in which these capabilities can be realised and JBossESB does not mandate one implementation over another. Therefore, all capabilities will be accesses as services which will give plug-and-play configurability and extensibility options.

## 2.1 ESB architecture requirements

In a distributed environment services can communicate with each other using a variety of message passing protocols. With the aid of client and server stub code, RPC semantics can be used to maintain the abstraction of local procedure calls across address space boundaries. Client stub code is a local

proxy for the remote object, which is controlled by the corresponding server stub code. It is the responsibility of the client stub to marshal information which identifies the remote method and its parameters, transmit this information across the network to the object, receive the reply message, and unmarshal the reply to return to the invoker.

SOA does not imply a specific carrier protocol and neither does it imply RPC semantics (in fact, loose coupling of services forces developers into an asynchronous message passing pattern[1]). Therefore, multiple protocols should be supported simultaneously. In most cases, clients will know the communication protocol to use when interacting with a service; however, in some situations this may not be the case, and the communication stack may need to be assembled dynamically (via a hand-shake protocol, where the client stub may have to be dynamically constructed[2]).

At the core of JBossESB is a MOM, but this MOM must itself be abstract, in that it will not force us into just JMS or SOAP styles. For example, a pure-play Web Services deployment within the ESB *must* be supported and, in which case, if reliability is a requirement as well then something like WS-RX (Web Services Reliable Exchange) will be needed from the underlying MOM abstraction[3]. As such, the ESB will assume a single MOM abstraction, but the capabilities may be provided by multiple different implementations. This is further support for the notion of having multiple buses within the ESB (each bus may be controlled by a separate MOM implementation).

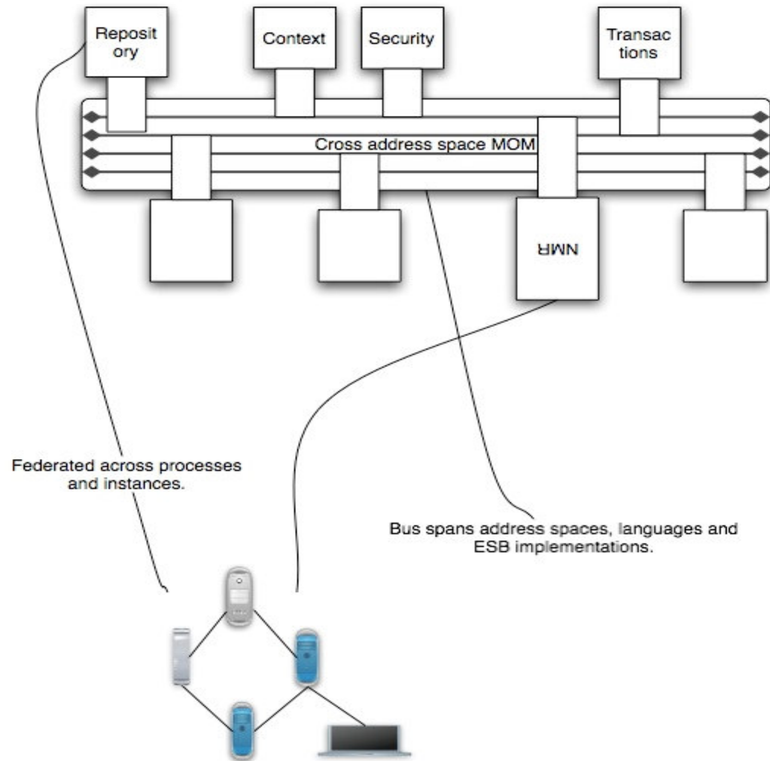
The service description and service contract are extremely important in the context of SOA and therefore ESB. In general, the developers create the contracts and the ESB maps it to whatever technology is being used to implement the SOA, e.g., WSDL. The JBossESB will allow this mapping to technology to be configurable and dynamic, i.e., it will support multiple SOA implementation technologies.

---

[1]Actually true asynchrony is often not necessary: synchronous one-way (void returns) RPCs can be used and often are in Web Services.

[2] Services may be available via multiple different protocols simultaneously, e.g., CORBA IIOP and JMS. A service repository (aka Name Service/Trading Service) will maintain service identities with their endpoint references and contract definitions (CORBA IDL, WSDL, etc.)

[3] Until sufficient capabilities exist in all possible implementation configurations, it may be that certain deployments will not be supported.




Using the ESB/SOA actually consists of two phases: the initial creation phase and the maintenance phase, which may have different requirements from the creation phase. Services evolve over time and it is often difficult or impossible to find a quiescent period in which to replace a service. As such, in any enterprise deployment there is likely going to be multiple versions of services being used by clients at the same time. Some of the version mismatch may be hidden by suitable routing and on-the-fly message modifications. JBossESB will address the challenge of versioning of services, something that other implementations tend to ignore. Preliminary thoughts are that services will be identifiable via major and minor version numbers, with pattern matching capabilities provided by a pluggable rules engine, e.g., a default rule would be that all minor versions are compatible within the scope of the same major version number, but that can be overridden with a specific rule by the service provider or system administrator.

There are actually two different aspects to the service bus: first, turning legacy systems and services into services that work within the SOA infrastructure; secondly, there is taking the services and adding policy and mediation control between those services. Integral to this is the notion of SOA Repositories: a repository is a persistent representation of an SOA Registry, which is needed to publish, discover and consume services. How registries/repositories are implemented is not important to the architecture: it's just another service after all. Example candidates would include UDDI implementations, or CORBA Trading Services. Therefore, the repository/registry needs to be in JBossESB from the start. Support for UDDI as one of the implementations will be included by default.

Nothing within the ESB should be a single point of failure. Inter-VM communication must be the default assumption.

Management, including monitoring and versioning of services, is important. There has been a lot of work in the Web Services arena on management and we should carefully examine this to determine its applicability.

## 2.2 Context service and sessions

The context service, as devised within WS-CAF and documented widely outside that technical committee, makes sense within an ESB/SOA framework: it is a critical component in ensuring that SOA applications can be developed that are loosely coupled.

One of the common features of all middleware systems is support for the *session concept*. A session is a mechanism for correlating multiple messages in order to achieve some application-visible semantic. This is typically done on behalf of a client within a service endpoint. In general, middleware systems decouple session association from specific communication channels to improve robustness. To achieve this, the session model is layered on top of a communication channel that links the client to network-visible application services. Many middleware systems advertise the session model explicitly as a mechanism for client applications to manage stateful conversations or communicate with stateful "resources". In other cases, the session concept is maintained less explicitly to support system services that are provided to applications.

At present there is no standard way of obtaining sessions in Web services and this is also a significant gap in SOA. However, there are two specifications that make proposals in this area, WS-Addressing and WS-Context. Although WS-Addressing is primarily concerned with identifying/addressing Web services (and hence in many ways is complimentary to WS-Context)[4], it provides a session-like concept as a by-product of the model, whereas WS-Context is primarily concerned with sessions. The WS-Addressing session model is based on experiences from previous closely-coupled middleware platforms such as CORBA and J2EE and does not work well in the loosely-coupled environment of SOA. The WS-Context approach is similar in many ways to the Cookie approach in the traditional Web and as such offers an approach that integrates well with the architecture.

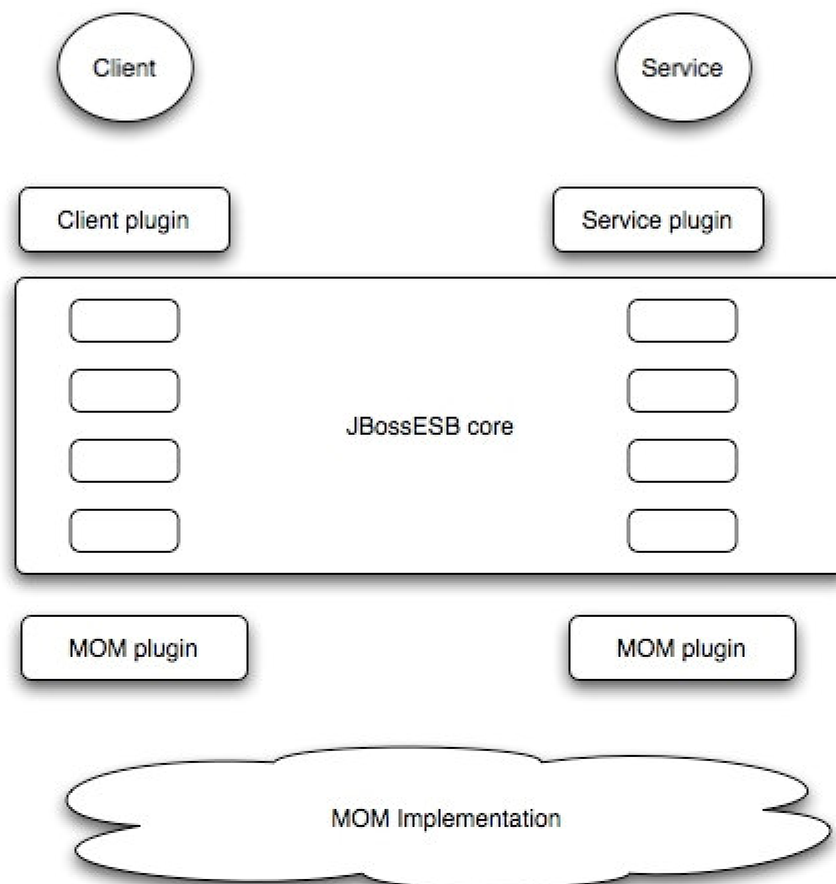
It may appear as though message-oriented middleware (MOM) systems only use channels to relay messages to queues or consumers – that any correlation semantic to backend state must be encoded in the message itself by applications. However, MOM systems offering message-grouping facilities can be applied to ordering and delivery assurance semantics. For example, in many MOM systems, a session is created to demarcate and manage the start of an ordered group. To end delivery of ordered messages within a group, the session is closed. As an example, the Web services standard WS-Reliability provides explicit protocol instructions to support this paradigm. Sessions are demarcated during application-level message exchange by implicit headers. Message acknowledgements may be communicated on independent communication channels, but messages are correlated with a group identifier that correlates messages with the session. While sessions in MOM systems are not necessarily explicitly accessed by message producers or consumers, the session concept is still very powerful and useful – including in systems that emphasize nominally decoupled message producers and consumers.

Supporting sessions in the SOA deployment environment is important. It has been shown that the WS-Context approach is superior to WS-Addressing's notion of session. As such, we shall be employing contexts such as those developed within WS-Context.

## 2.3 Implementation interfaces

A high-level architectural overview of JBossESB is shown below. In this section, we shall examine the various interfaces illustrated.

[4]Which is why a WS-Addressing-like address scheme still makes sense.




### 2.3.1 Addresses

Services can be addressed implicitly (via logical names) as well as explicitly (endpoint references, or EPRs). Sending and receiving of messages require styles such as those typified in WSA, e.g., wsa:To, wsa:ReplyTo and wsa:From. We shall use the WS-Addressing EPR format for service identification.

- Logical service names require mapping to address(-es) via name service/registry. A federated approach will be used, to allow registries to span multiple repositories and implementations.
- Address-based approach based on the wsa:EPR format.

The addressing scheme should not be tied to a specific delivery pattern or implementation.

Logical names will be resolved via the repository; this may also be used to provide the basis of the mediation/indirection mechanism of traditional ESBs: client talks to service A via a logical name LogicalAName; the mapping of LogicalAName to physical EPR produces mediation service EPR with ReferenceParameter including service A EPR (or some means for the mediation service to determine the ultimate destination service EPR).

```
interface EPR
{
	public void setTo (URI uri);
	public URI getTo ();
	public void setFrom (URI uri);
	public URI getFrom ();
	public void setReplyTo (URI uri);
	public URI getReplyTo ();
	public void setFaultTo (URI uri);
	public URI getFaultTo ();
	public void setAction (URI uri);
	public URI getAction ();
	public void setMessageID (URI uri);
	public URI getMessageID ();
	public void setMetaData (MetaData md);
	public MetaData getMetaData ();
	public void addReferenceParameter (...);
}
```

### 2.3.2 The MOM abstraction

The core of JBossESB is a MOM abstraction which supports multiple implementations. Most remote communication mechanisms expose their functionality in their interfaces, e.g., by requiring users specify timeout and retry values when constructing the message to be sent. Therefore, designing a MOM abstraction which (dynamically) supports multiple implementations requires selecting an appropriate interface that does not implicitly bind an application to a specific implementation.

However, before looking at the MOM interfaces, we shall first examine the message. There are two representations of the message:

- The message that the client/service uses, which is XML based, but the structure of which is outside the scope of JBossESB.
- The message the ESB core uses internally, which is also XML based.

The core ESB message consists of a header, a context and a body, and is shown below:

```
interface Message
{
	public Header getHeader ();
	public Context getContext ();
	public Body getBody ();
	public Fault getFault ();
	public Attachment getAttachment ();
}
```

The header consists of unique message identifier and any necessary routing information. The context consists of information such as transactions, security, etc. Clients and services work in terms of the XML Body component of this message. The rest of the Message is formed by a combination of the plugins and the ESB core.

The MOM abstraction used by JBossESB is broken down into two boundaries as illustrated in the previous figure:

- The client/service plugin; as mentioned above, this, in conjunction with service requirements, is responsible for the initial formation of the Message, given as input the payload and destination address/name.
- The MOM plugin; this takes the Message and utilises the information is contains to route the payload to the ultimate destination.

The client interface is shown below:

```
interface ClientPlugin
{
	public void send (Address to, Body msg);
	public void sendAsync (Address to, Body msg);
	public void sendAsync (Address to, Body msg, Callback cb);
	public void sendReliable (Address to, Body msg);
}
```

The client plugin is obtained from a factory:

```
interface ClientPluginFactory
{
	public ClientPlugin getPlugin (ContractDefinition def);
}
```

The ContractDefinition (to be defined) is used to define QoS requirements for the plugin, e.g., reliability level, messaging implementation etc. With this abstraction, a single instance may be used by clients to interact with an arbitrary number of services. Furthermore, the instance is effectively stateless and could be used when interacting with a service multiple times within the same session.

Because the core architectural component within JBossESB is the message, the service-side interface is relatively straightforward:

```
interface ServicePlugin
{
	public Body receive (Address from);
}
```

This is also obtained from a factory:

```
interface ServicePluginFactory
{
	public ServicePlugin getPlugin (ContractDefinition def);
}
```

Within the JBossESB core (between client/service and MOM implementation), we will be exploiting the Dispatcher interface for components, which works in terms of Messages:

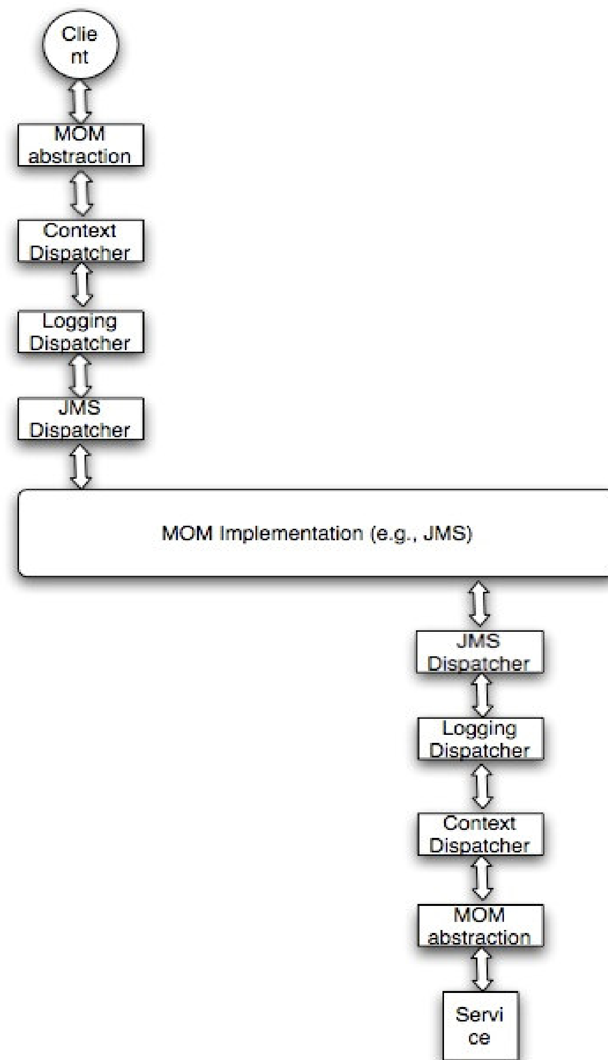
```
interface Dispatcher
{
    public Message[] dispatch (Message[] msg);
}
```

Dispatchers may be stateful and represent a specific service(s). This allows for intelligent and adaptable implementation hierarchies to be built up over time - important for long duration interactions, particularly when the underlying MOM implementation may not support all required functionality. For example, retained results: in this case, lost response messages and request retry attempts may cause the same operation to be performed by a service when that operation is not idempotent (e.g., transfer $X to account Y); with retained results, responses are held at the server side and replayed from a lower-level in the hierarchy if a retry is observed, i.e., the server logic does not see retries[5].

As we shall see, a uniform interface such as Dispatcher, which also does not expose any directionality to messages, allows us to support dynamic insertion of capabilities without requiring changes to the application code.

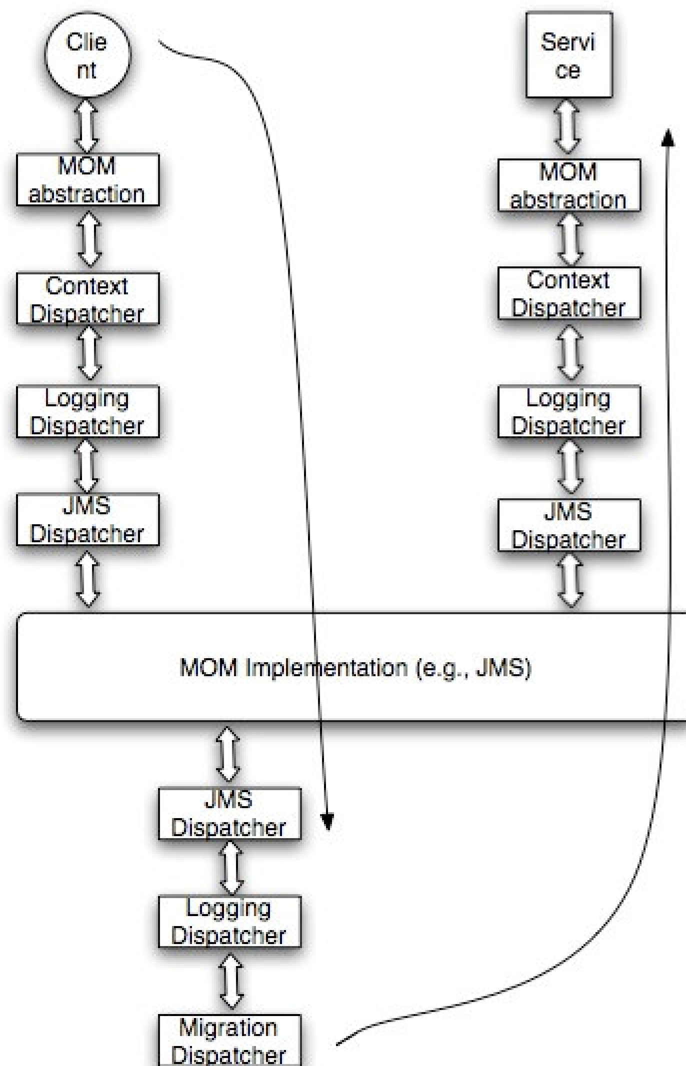
Users call the `dispatch` method with an array of work `Message` and, in a synchronous request/response MEP, expect to be returned an array of result `Message`. The interface conveys no information about how the implementation generates the results from the work: in fact no network communication need be involved at all. Implementation specific information, such as why failures occurred, is encoded in the Message.

[5] An appropriate set of dispatchers can be used in conjunction with a MOM implementation to provide client or service functionality that may not be provided directly by the MOM.




Using this model, client and services are represented as dispatcher hierarchies. These hierarchies are configured when created, through an appropriate XML document and can be reconfigured dynamically.

Because each layer interacts with its neighbours through the `Dispatcher` interface, it is straightforward to replace/insert/remove a layer without requiring a recompilation of the application. In addition, this reconfiguration can occur dynamically as the application executes. For example, it would be possible to reconfigure the system to have an additional *Logging Dispatcher*, which is responsible for logging the progress of the distributed interactions, and possibly displaying them graphically.




For example, in the diagram above, we can support dynamic service migration by replacing one portion of an existing service dispatcher hierarchy with a Migration Dispatcher (essentially a forwarding address). Some MOM implementations may support migration natively, but in the situations where this is not the case, we can provide the necessary support. Multiple migrations are possible in this case, with short-cutting of chains handled automatically by the infrastructure.

The interface between Dispatchers and the MOM implementation is provided by the ClientMOMPlugin or ServiceMOMPlugin at the client and service side respectively:

```
interface ClientMOMPlugin
{
	public void send (Message msg);
	public void sendAsync (Message msg);
	public void sendAsync (Message msg, Callback cb);
	public void sendReliable (Message msg);
}
interface ServiceMOMPlugin
{
	public Message receive ();
}
```

## 2.4 Contract definitions, policies, governance and service behaviour

Contract definitions for services, indicating QoS requirements, will be supported. Whatever is used must map to WS-Policy. Furthermore, because the contract definition is used to bind the client/service

to the bus, it must support sufficient syntax to provide capabilities such as binding to multiple buses (equivalent to publish/subscribe on multiple topics in JMS).

Component interface defines getPolicies(). JBossESB will provide an appropriate Policy interface.

WSDL should not be the contract definition language – unless it can be automatically generated from something far simpler. WSDL is too Web Services specific: we shall use something more generically SOA.

Services implement a basic set of management operations.

SOA governance is important for the ESB design.

WS-CDL as a possible language for expressing the behaviour of the entire system.