Enterprise Service Bus

Falko Menge

***Abstract*— This paper is a comprehensive introduction to the Enterprise Service Bus (ESB), which is a new type of integration infrastructure. Therefore it gives a detailed introduction into the background of Enterprise Application Integration (EAI) and explains Message-Oriented Middleware (MOM) and Service-Oriented Architecture (SOA) which are the technologies the Enterprise Service Bus evolved out of. After discussing core ESB concepts and typical features, the open source ESB solution Mule will be presented. An example application will demonstrate how different features of an ESB are used.**

***Index Terms*— Enterprise Service Bus (ESB), Service-Oriented Architecture (SOA), Web Services, Enterprise Application Integration (EAI), Message-Oriented Middleware (MOM).**

I. Introduction

THE Enterprise Service Bus (ESB) is a relatively young term in the software industry. It is an exciting topic because the most interesting question is the first, one would ask: What is an Enterprise Service Bus?

The question is hard to answer since there is no general consensus about a common definition of the term. There are many discussions on which features have to be included or which technologies should be used when realizing an Enterprise Service Bus. In contrast to that there are many vendors in the market who state that their solutions are Enterprise Service Buses or base on Enterprise Service Bus principles. There are vendors who are specialized in developing Enterprise Service Bus products like Cape Clear Software, Fiorano Software or Sonic Software and most of the big vendors of integration middleware products like Oracle, BEA Systems, IONA Technologies, Sun Microsystems or IBM have Enterprise Service Bus solutions in their product portfolios.

This uncommon situation originates from the term Enterprise Service Bus being originally coined by analysts from Gartner in 2002. The need for a new form of infrastructure which combines Message-Oriented Middleware (MOM), web services, transformation and routing intelligence as a backbone for Service-Oriented Architecture (SOA) was identified and different approaches have been tried to realize that idea. Some products evolved from web services infrastructure solutions or lightweight messaging products, while others evolved from EAI suites having added support for SOA.

Although there are multiple definitions and approaches, the central ideas are similar to each other and will be explained and demonstrated in this paper.

II. Background and Domain

In order to understand the benefits of an Enterprise Service Bus one has to examine the infrastructure requirements that large enterprises have. A typical scenario is that an enterprise runs hundreds or thousands of applications, which could be custom built, acquired from a third party or parts of legacy systems, e.g. a company may have three installations of SAP, 30 different websites and countless individual solutions in different departments. These applications should be able to communicate and exchange data with each other in order to work together for the business of the company. The problem is not the number of applications itself. There are good reasons for having combinations of many different applications. First, it is nearly impossible to develop one huge application which performs all business functions of a typical enterprise because there are far too many requirements. The second reason is that running multiple applications gives IT managers flexibility to select solutions which are the best for their particular purpose. That means that integration is not a temporary problem which currently has to be solved. It is a fundamental requirement that enterprises will also have in the future. The task, commonly called Enterprise Application Integration (EAI), becomes even more interesting if applications of external business partners are to be integrated. Furthermore the selection of applications is not fixed. Applications may be exchanged or new applications may be deployed so that the integration infrastructure has to be able to handle new scenarios.

A. Message-Oriented Middleware

The traditional solution for Enterprise Application Integration is to use Message-Oriented Middleware (MOM). That means that asynchronous messaging is used to decouple the applications from each other. MOM products are typically built around a central message queue system often called message broker. All applications are connected to the message broker using a unified interface for sending and receiving messages.

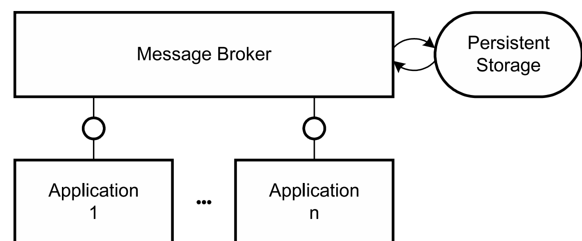

Fig. 1. Simplified architecture of a Message-Oriented Middleware.

The message broker is able to store the messages so that sender and receiver do not need to be connected at the same time. Within this middleware layer messages can be routed which makes it possible to deliver a single message to more than one recipient. Furthermore messages cold be transformed by the message broker to fit the requirements of the receiving application. The transformation facilities allow the connected applications to use their own native message formats.

A big problem of MOM solutions is that they often use proprietary protocols and platform specific interfaces and deployments. This leads to a total dependency of the applications on the infrastructure and causes interoperability problems with MOM products of alternative vendors. As a result islands of MOM based infrastructures can often be found.

B. Service-Oriented Architecture

Service-Oriented Architecture (SOA) is an architecture concept which defines that applications provide their business functionality in the form of reusable services. A service in that context is a self-contained and stateless business function which is accessible through a standardized, implementation-neutral interface. Services are used by other applications which could also be implementations of services. With this approach complex business processes are implemented through combination of several services. This is called service orchestration. Figure 2 shows the typical scenario of a SOA application. Service providers register their services to a central naming service. A consumer application can use this naming service to discover available services and retrieve information on how to connect to a particular service provider. Then the consumer application is able to obtain a service description which defines how the service can be used.

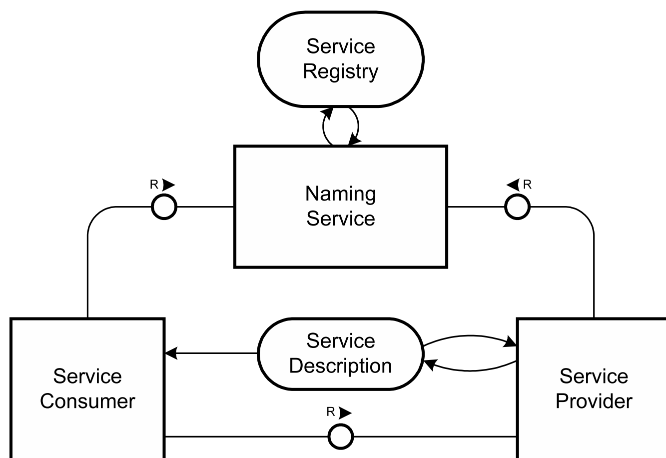

Fig. 2. A typical Service-Oriented Architecture.

SOA can be implemented using any service-based technology. Typically web service technologies like SOAP or REST are used.

SOA allows complex enterprise applications and end-to-end business processes to be composed from these services, even when the providers of those services are applications hosted on disparate operating system platforms, written in different programming languages or based on separate data models. This flexible composition supports the fundamental goals of business integration, which are linking business systems across the enterprise and extending business services to customers and trading partners.

The adoption of SOA in business-critical applications is occurring only incrementally. Refactoring, wrappering or replacing legacy applications with new standards-aware equivalents is due to be a slow process. That implies that an integration infrastructure can not be purely service-based.

Today enterprises need powerful integration solutions but they want them to be based on open standards and to support Service-Oriented Architecture. Exactly those requirements led to the idea of an Enterprise Service Bus.

III. What is an Enterprise Service Bus?

An Enterprise Service Bus is an open standards, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services in a secure and reliable manner.

ESBs are usually realized through service containers distributed across a networked environment. These containers host integration services like routers, transformers, application adapters or MOM bridges and provide them with a broad range of communication facilities. In today's ESB solutions the messaging infrastructure is typically built on top of JMS-based[1] middleware systems which guarantee message delivery. Applications are connected to the bus using application adapters or one of the supported messaging mechanisms. In order to support SOA the ESB service containers have to include all important web service technologies. The components of the ESB as well as the mechanisms for connecting resources must be based on open standards to ensure interoperability and protection of investment.

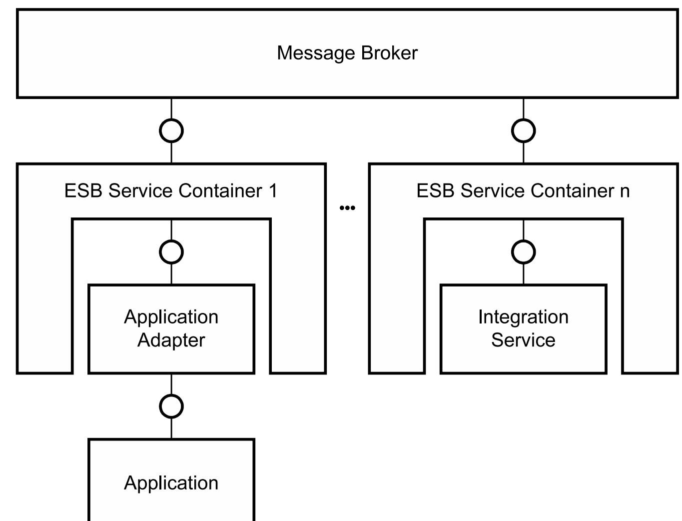

Fig. 3. A simple ESB scenario. Although not depicted here, a container can host multiple services and components.

The ESB must coordinate the interactions of the various resources and provide transactional support. A general goal is to provide messaging and integration without writing code. Therefore generic components are provided which can be configured to realize a desired scenario.

An ESB is an ideal backbone for implementing service oriented architectures because it provides universal mechanism

[1]Java Messaging Service (JMS) specifies a standardized API for reliable messaging. JMS is widely supported in messaging middleware systems and is developed as JSR 914 [12]

to interconnect all the services required in the composed business solution without compromising security, reliability, performance and scalability.

IV. Typical ESB features

A. Invocation

Invocation is the ability of an ESB to send requests and receive responses from integration services and integrated resources. That means that an ESB has to support the standards for web service communication including SOAP [14], the Web Services Description Language (WSDL) [15], Universal Description, Discovery and Integration (UDDI) [16] and the WS-* family of standards. Furthermore the Java Message Service (JMS) API [12] and the J2EE Connector Architecture (JCA) [13] should be implemented for integration with MOM systems and application servers. Of course an ESB must also be able to handle the underlying protocols like TCP, UDP, HTTP or SSL. Additional communication mechanisms could be JBI, RMI, JDBC, SMTP, POP3, FTP or XMPP.

B. Routing

Routing is the ability to decide the destination of a message during its transport. Routing services are an essential feature of an ESB because they allow to decouple the source of a message from the ultimate destination.

To enable routing and other communication features disparate messaging endpoints have to be referenced. The common standard for addressing is to use Uniform Resource Identifiers (URIs). Additionally WS-Addressing [17] may be implemented in ESB solutions to describe web service endpoints in a transport-neutral manner.

The decision to which destination a message is sent can be made based on several conditions which leads to different types of routers.

A content-based router inspects the content of messages and forwards them to different channels depending on the content of the message. This allows a sender to send messages without specifying an exact destination. For XML messages content-based routing can be implemented using the XML Path Language (XPath) [19] for addressing parts of a message which are needed for the routing decision. A special form of a content-based router is a message filter. It forwards a message only if the content matches a certain criterion. Otherwise, the message will be deleted.

Of course there are integration scenarios where one message is to be sent to multiple recipients. This could be achieved by using a recipient list router which either computes a list of destinations or has a static configuration for multiple receivers. If messages are composed of multiple parts, a splitter may be used to split a single message into a series of individual messages. A splitter for XML messages may utilize XPath for addressing the parts and transformations based on the Extensible Stylesheet Language (XSL) [18] for generating separate messages.

The opposite of a splitter is an aggregator which collects and stores individual messages. If it receives a complete set of related messages, the aggregator sends a single message distilled from the individual messages to the configured destination. A resequencer is a router which collects related but out-of-sequence messages and forwards them in the correct order. For the resequencer to function, each message needs a unique sequence number.

The different routers can be combined to create complex message flows. All routers can be implemented as so called dynamic routers. That means the router is able to reconfigure its routing rules based on special configuration messages which can be sent by participating destinations.

C. Mediation

Mediation refers to all transformations or translations between disparate resources including transport protocol, message format and message content. These transformations are very important for integration because applications rarely agree on a common data format. Again XSL and XPath are powerful tools for working with XML messages. Those standards allow an ESB to provide generic XML transformer components which are configured through an XSL stylesheet. More complex transformers may invoke other resources which are connected to the bus e.g. a database in order to augment additional information to a message. Special forms of transformation services are normalizers, content enrichers, content filters or envelope wrappers.

D. Adapters

Many ESB solutions provide a whole range of application adapters. This can be adapters for popular application packages such as Enterprise Resource Planning (ERP), Supply Chain Management (SCM) and Customer Relationship Management (CRM). Those adapters connect to the native transaction interfaces, APIs and data structures that these business applications expose and present a standard interface, which makes it easy to reuse business logic and data. Typically most adapters of a particular vendor operate the same way which minimizes the skills required to use each connected system. Using prefabricated adapters reduces the work required to integrate applications into a Service-Oriented Architecture.

E. Security

An enterprise class integration infrastructure has to provide secure messaging. That means for an ESB to be able to encrypt and decrypt the content of messages, handle authentication and access control for messaging endpoints and to use secure persistence mechanisms.

F. Management

An ESB has to provide audit and logging facilities for monitoring infrastructure and integration scenario and possibly also for controlling process execution. There must be a central mechanism for configuration and administration of the bus. Additionally tools for usage metering may be included.

G. Process Orchestration

An Enterprise Service Bus may include an engine to execute business processes described with the Web Services Business Process Execution Language (WS-BPEL) [11]. This engine controlled by the process description then coordinates the collaboration of the services connected to the bus.

H. Complex Event Processing

An asynchronous message can be seen as an event especially when using a publish-subscribe channel. Thus an ESB may include mechanisms for event interpretation, event correlation and event pattern matching which enable event-driven architectures.

I. Integration Tooling

For professional development with an ESB graphical design-time tooling as well as deployment and testing tool should be available.

V. Mule: an open source ESB

Mule is a lightweight open source messaging platform based on ESB concepts. The central component of Mule is a service container for so called Universal Message Objects (UMOs). This container which is a highly distributable object broker is able to handle a rich variety of communication mechanisms including JMS, SOAP, REST, HTTP, FTP, TCP, UDP, SSL, XMPP, SMTP, POP3, IMAP, JDBC, RMI and others to come.

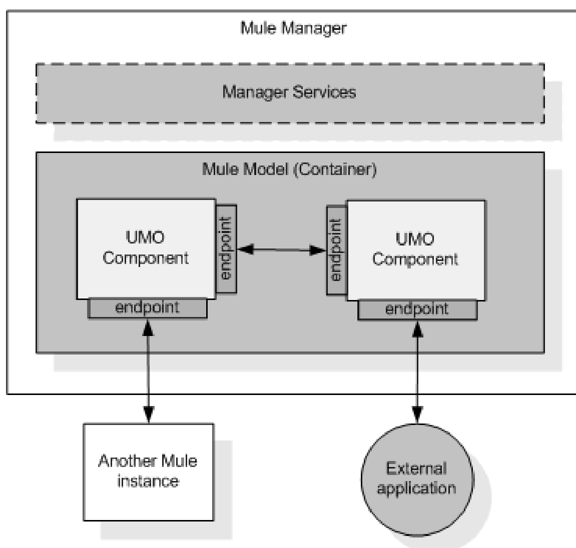

Fig. 4. Simplified view of a Mule instance. [source: [10]]

The Universal Message Objects are plain old Java objects which can transparently use the communication facilities of the container trough a generic message endpoint interface. As shown in figure 4 they are able to connect to external applications, other UMOs or other Mule instances. This unified and technology independent method for interacting with disparate resources makes it very easy to use UMOs for integrating applications with each other or implementing own application adapters.

A typical Mule application consists of multiple Mule instances distributed across the network which can be interconnected using any of the supported communication mechanisms or even a combination of them. Several UMOs can work together in oder to realize a complex message flow among different external applications.

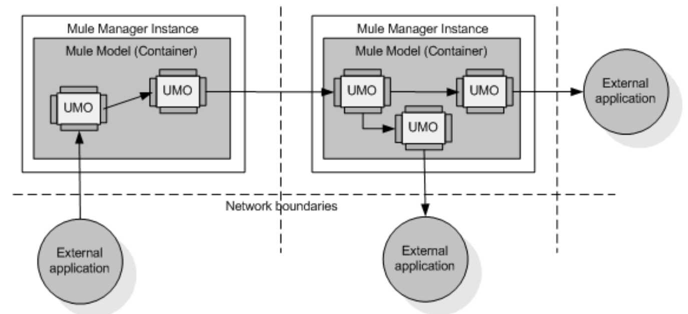

Fig. 5. A typical Mule application. [source: [10]]

The Mule container provides all integration services which are essential to an ESB. That includes support for transactions, transformations, routing, logging, auditing, management, security, event processing and even process orchestration using WS-BPEL. Mule can run standalone or integrated with another containers, e.g. Spring, PicoContainer or Plexus. Therefore it separates object construction from management so that service objects can be constructed by other containers. Mule avoids to reimplement communication technologies and instead uses stable and widely-accepted implementations, e.g. the Axis and Glue SOAP stacks.

The simple integration scenario in figure 6 shows the message flow between two applications through a UMO component. A message receiver listens on the incoming endpoint for a message from the sending application. The connector handles the particular transport mechanism and after that a transformer may transform the message, e.g. into a Java object. The so called inbound router of the involved UMO may be a message filter, aggregator or resequencer. The UMO may perform some business logic on the message and then the outbound router which could be a content-based router, message filter, splitter or recipient list forwards the message to one or more transport providers which again consist of a transformer, connector and in the outgoing case a message dispatcher. The message dispatcher sends the message to the receiving application.

Mule is designed for fast development of distributed service networks. Therefore it is an ESB server which is highly scalable, lightweight, fast and simple to adopt. Mule allows smaller projects to leverage enterprise level service architectures, especially when resources, development cost and TTM need to be kept to a minimum.

VI. An ESB Example Application with Mule

An important principle of an ESB is to use as much declaration and configuration as possible and avoid writing code. Thus the most important artifact of a mule application is its XML configuration file in which the connectors, routers,

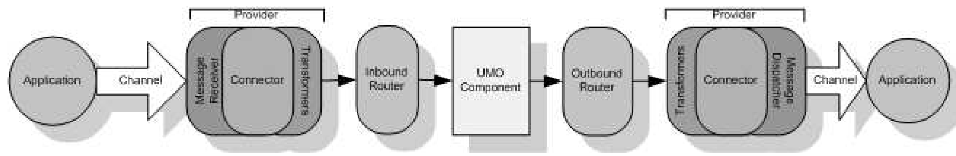

Fig. 6. Integration of two applications with Mule. [source: [10]]

transformers, message endpoints and UMOs are declared. The communication mechanisms actually used by and application are simply configured through the endpoint URIs. In addition to that only UMOs and maybe own object transformers or routers have to be programmed.

The sample application which will be discussed in this section is a very popular example taken from the Enterprise Integration Patterns book [1]. The example is often used to demonstrate features of messaging systems and so Ross Mason implemented the application in different flavors for Mule. The version shown here uses an ESB architecture and can be obtained from the mule website [10].

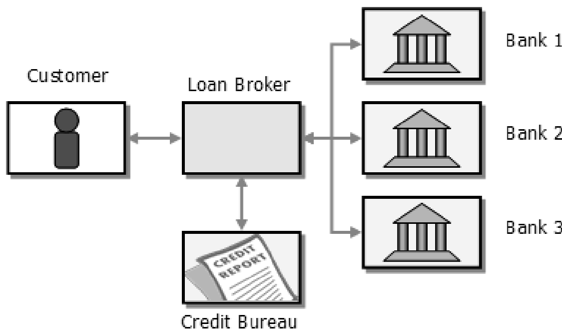

Fig. 7. Mule example application: A loan broker acting as intermediary for a consumer talking to banks in order to get a loan quote. [source: [1]]

The usage scenario is a customer who wants to obtain a loan quote. Therefore he wants to compare offers from different banks. A loan broker is used to communicate with the banks and send the best offer back to the customer.

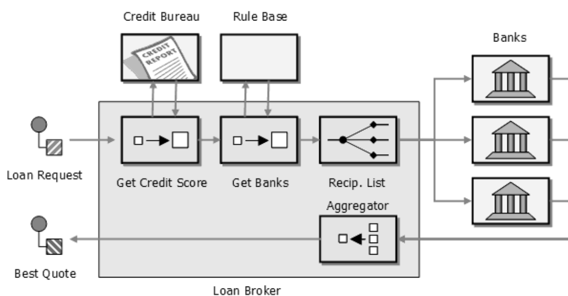

Fig. 8. Mule example application: Design of the application. [source: [1]]

As seen in figure 9 the sample application uses a JMS message broker to connect the different components on the bus, which is cofigured through the following connector deklaration:

```
<connector name="jmsConnector"
 className="org.mule.providers.jms.JmsConnector">
  <properties>
    <property name="connectionFactoryJndiName"
     value="ConnectionFactory"/>
    <property name="jndiInitialFactory"
     value=
"org.activemq.jndi.ActiveMQInitialContextFactory"
     />
    <property name="specification" value="1.1"/>
    <map name="jndiProviderProperties">
      <property name="brokerURL"
       value="tcp://localhost:61616"/>
    </map>
</properties>
</connector>
```

At the beginning the Client application makes a request sending a CustomerQuoteRequest message to the LoanBroker via a RESTful web service. The LoanBroker which is a UMO creates a LoanQuoteRequest message which is the common message format on the bus. The following part of the configuration shows how different message endpoints are attached to the LoanBroaker UMO:

```
<mule-descriptor name="LoanBroker"
 implementation=
 "org.mule.samples.loanbroker.esb.LoanBroker  ">

  <inbound-router>
    <endpoint
     address="LoanBrokerRequestsREST"
     transformers="RestRequestToCustomerRequest"/>
    <endpoint
     address="LoanBrokerRequests"/>
  </inbound-router>

  <outbound-router>
    <router className=
"org.mule.routing.outbound.OutboundPassThroughRouter">
      <endpoint address="CreditAgencyGateway"/>
    </router>
  </outbound-router>

  <response-router timeout="1000000">
    <endpoint address="LoanQuotes"/>
    <router className=
"org.mule.samples.loanbroker.esb
.routers.BankQuotesResponseAggregator"
     />
  </response-router>
</mule-descriptor>
```

Mule sends the message to the so called Credit Agency Gateway via JMS. The Gateway is an envelope wrapper which marshals the request and invokes a CreditAgency EJB. A prefabricated message enricher component called Reflection-MessageBuilder automatically attaches a CreditProfile to the LoanQuoteRequest message. Then Mule sends the Message to the Lender Gateway via JMS. This Gateway uses the VM transport to invoke the Lender Application which is a simple

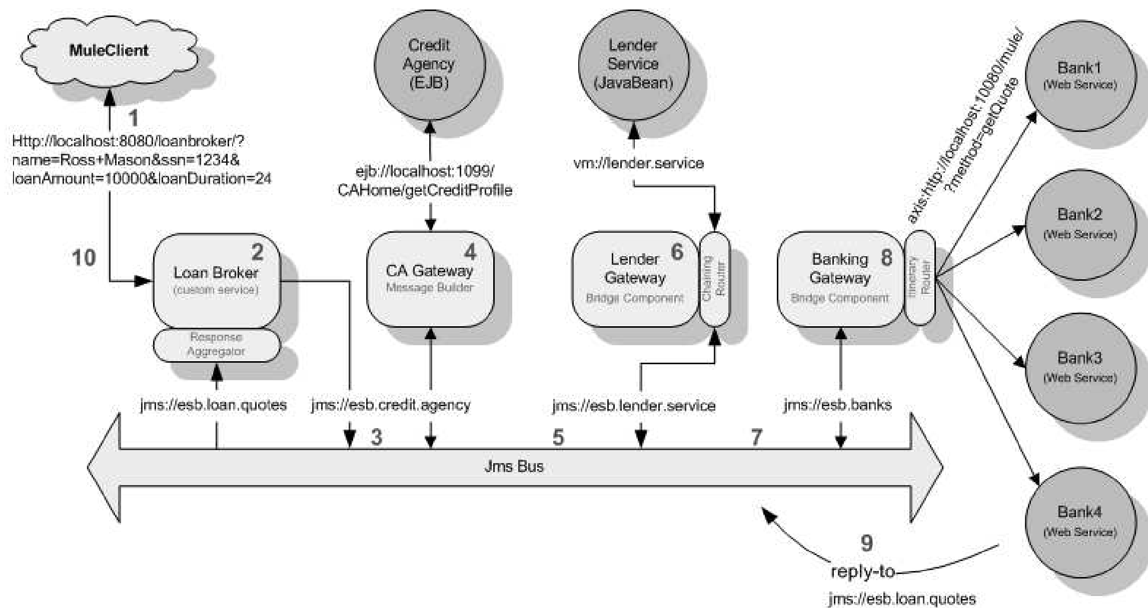

Fig. 9. Mule example application: LoanBroker ESB. [source: [10]]

Java Bean. Again Mule forwards the message via JMS - now to the Banking Gateway. The Banking Gateway realizes the Scatter-Gather integration pattern [1]. Based on a recipient list a router forwards a request via SOAP to several bank web services. The response messages are collected by an aggregator specified in the ReplyTo address provided by the Banking Gateway. The ResponseRouter on the LoanBroker UMO receives the responses. Finally the LoanBroker selects the lowest quote received for the request and returns it to the client.

VII. Conclusion

An Enterprise Service Bus (ESB) is a message based, distributed integration solution which provides integration services. Basic integration services are routing, invocation, mediation, support for transactions, logging, auditing, management and security services. Furthermore event processing and process orchestration may be supported by an ESB.

Today an Enterprise Service Bus is the most popular infrastructure for implementing Service-Oriented Architectures.

Mule as a light-weight open source ESB includes the most important features and is an ideal entry point for ESB adoption.

References

[1] Gregor Hohpe and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions* Pearson Education, Inc., 2004.
[2] *Architectural Overview for Enterprise Integration*, Version 1.0 Cape Clear Software, March 2003.
[3] Dirk Krafzig, Karl Banke and Dirk Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices* Prentice Hall PTR, 2004.
[4] Tony Baer, *The road to SOA* CBR Research, 2006.
[5] David A. Chappell, *Enterprise Service Bus* O'Reilly, 2004.
[6] Mike Gilpin and Ken Vollmer, *The Forrester Wave: Enterprise Service Bus, Q4 2005* Forrester Research, Inc., November 15, 2005.
[7] *Cape Clear's Enterprise Service Bus (ESB)*, Whitepaper Cape Clear Software, May 2005.
[8] *Principles of BPEL, Orchestration, and the ESB*, Whitepaper Cape Clear Software, 2004.
[9] Bruce Silver *Enterprise Service Bus Technology for Real-World Solutions* Bruce Silver Associates, August 2004.
[10] Official website of the Mule project. http://mule.codehaus.org
[11] *Web Services Business Process Execution Language (WS-BPEL) Version 2.0* OASIS, Decmber 21, 2005. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm
[12] *JSR 914: Java Message Service (JMS) API* http://www.jcp.org/en/jsr/detail?id=914
[13] *JSR 112: J2EE Connector Architecture 1.5* http://www.jcp.org/en/jsr/detail?id=112
[14] *SOAP Version 1.2 specification*, W3C Recommendation World Wide Web Consortium (W3C), June 24, 2003. http://www.w3.org/TR/soap/
[15] *Web Services Description Language (WSDL) 1.1*, W3C Note World Wide Web Consortium (W3C), March 15, 2001. http://www.w3.org/TR/wsdl/
[16] *Universal Description, Discovery and Integration v3.0.2 (UDDI)* OASIS, Oktober 19, 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm
[17] *Web Services Addressing (WS-Addressing)*, W3C Member Submission World Wide Web Consortium (W3C), August 10, 2004. http://www.w3.org/Submission/ws-addressing/
[18] *Extensible Stylesheet Language (XSL) Version 1.1*, W3C Candidate Recommendation World Wide Web Consortium (W3C), February 20, 2006. http://www.w3.org/TR/xsl11/
[19] *XML Path Language (XPath)*, W3C Recommendation World Wide Web Consortium (W3C), November 16, 1999. http://www.w3.org/Submission/ws-addressing/