# An Introduction to the Enterprise Service Bus

Martin Breest

Hasso-Plattner-Institute for IT Systems Engineering at the University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
martin.breest@student.hpi.uni-potsdam.de

**Abstract.** The enterprise service bus (ESB) is the most promising approach to enterprise application integration (EAI) of the last years. It promises to build up a service-oriented architecture (SOA) by iteratively integrating all kinds of isolated applications into a decentralized infrastructure. This infrastructure combines best practices from EAI, like message-oriented middleware (MOM), (Web) services, routing and XML processing facilities, in order to provide, use and compose (Web) services. Because the term ESB is often used to name different things, for example an architecture, a product or a "way of doing things", I point out in this paper what exactly an ESB is. Therefore, I first describe what distinguishes the ESB from former EAI solutions. Second, I show what the key components of an ESB are. Finally, I explain how these key components function alone and how they work together to achieve the aforementioned goal.

## 1 Introduction

Due to the ongoing globalization, enterprises all over the world have to face a fierce competition. In order to stay in business, they constantly have to automate their business processes, integrate with their business partners and provide new services to their customers.

With the changing demands in business, the goal of IT has also changed. Today, IT has to actively support enterprises in global competition. Therefore, it has to make business functionality and information available across the enterprise in order to allow software engineers to create, automate and integrate business processes and company workers to access all kinds of information in a unified way via a department- or enterprise-wide portal.

Today, most companies try to achieve the aforementioned goal by developing a service-oriented architecture (SOA) [4, 5]. In a SOA, the business functionality, implemented by different applications in the enterprise, is provided via coarse-grained, loosely-coupled business services. These business services allow to easily create and automate business processes by using, reusing and composing the provided business functionality.

The enterprise service bus (ESB) [1, 8] promises to build up a SOA by iteratively integrating isolated applications into a decentralized infrastructure. Various research and consulting companies like Forrester Research, IDC or Gartner

Inc. believe that ESB is the most promising approach for enterprise application integration (EAI) [2, 3] of the last years. Forrester Research for example regards the ESB as "a layer of middleware through which a set of core (reusable) business services are made widely available". IDC believes that "the ESB will revolutionize IT and enable flexible and scalable distributed computing for generations to come". Gartner Inc. analyst Roy Schulte wrote in 2002: "A new form of enterprise service bus (ESB) infrastructure, combining message-oriented middleware, Web services, transformation and routing intelligence, will be running in the majority of enterprises by 2005. ... These high-function, low-cost ESBs are well suited to be the backbone for service-oriented architectures and the enterprise nervous system".

Because the term ESB is obviously not clearly defined and often used to name different things, for example an architecture, a product or a "way of doing things", I point out in this paper what exactly an ESB is. Therefore, in section 2, I discuss what the difference between ESB and former EAI solutions is. In section 3, I describe what the key components of an ESB are and how they work together. In this section, I also explain the most important features of the three components message-oriented middleware (MOM), service container and management facility in detail. In section 4, I describe the special facilities of an ESB, which are the routing and XML processing facilities. Finally, in the last section, I give a conclusion and a short and final answer to the most important question: "What is (an) ESB?".

Throughout this paper, I use the block diagram notation of the fundamental modeling concepts (FMC) [7, 15] to illustrate architectural issues and the business process modeling notation (BPMN) [14] to illustrate business process issues.

## 2 The ESB: An Innovative Approach to EAI

ESB is about enterprise application integration. Whether the ESB approach to integration is innovative or not is open for discussion. However, as a matter of fact, most enterprises today try to develop a SOA by using an ESB. Because of that, I introduce my work by answering the following questions: "Why do we need Integration?", "Why do we need the ESB?" and "What does the ESB promise?".

### 2.1 Why do we need Integration?

The IT landscape that we find in most enterprises today, is a result of a historical development with a missing long-term strategy. It emerged from different IT projects that have been conducted to develop new applications, to refactor existing applications or to buy, customize and introduce standard applications.

The result of this development is a heterogeneous IT landscape that consists of a variety of different applications. Each of these applications has been bought

for a particular purpose, supports people in a specific domain and is owned by a certain department in the enterprise.

Naturally, the heads of the departments try to protect their resources, which are in our case the machines and applications that they bought from their budget, and the information gathered and maintained by their people. Therefore, they only share their resources if it is either beneficial for themself or if the enterprise's management forces them to do so. The result of this behaviour is that the IT landscape inside a department and across the enterprise often consists of many isolated applications.

Due to the ongoing globalization, enterprises today have to face a fierce competition. To stay in business the have to reduce their costs through process optimizations and gain new market shares through process and product innovations. Therefore, today's IT has to actively support enterprises in their development by continuously automating processes, integrating with business partners and delivering new business services to customers. In order to achieve this, applications from different domains and departments have to be integrated. As a consequence, to keep their departments alive and to not offend the enterprise's management to much, the heads of different departments have to start collaborating.

Collaboration happens in those cases where two or more heads of a department agree, after tough negotiations, upon sharing a certain piece of information or a specific business functionality. To actually integrate their applications, they setup one or more integration projects. Each integration project has the goal to integrate the affected applications.

The two common approaches for application integration in the past have been point-to-point integration and integration using a centralized EAI broker. A point-to-point integration aims at directly connecting two applications. An integration using an EAI broker has the goal to connect two or more applications via a centralized mediator. This mediator is capable of routing and transforming messages sent between the applications.

### 2.2 Why do we need the ESB?

The result of conducting numerous point-to-point and EAI integration projects is the so called accidental architecture. It consists on the one hand of unreliable point-to-point connections between tightly coupled applications and on the other hand of so called islands of integration.

The point-to-point integration approach leads to unreliable, insecure, non-monitorable and in general non-manageable communication channels between applications. The problem of this approach is also that the applications are tightly-coupled, which means that the integrating application has to know the target application, the interface methods to call, the required protocol to talk and the required data format to send. The general problem is that process and data transformation logic are encoded into the applications. Thus, each time a change occurs in an application, a new integration project has to be launched in order to refactor the depending applications. Figure 1 illustrates an example of an accidental architecture.

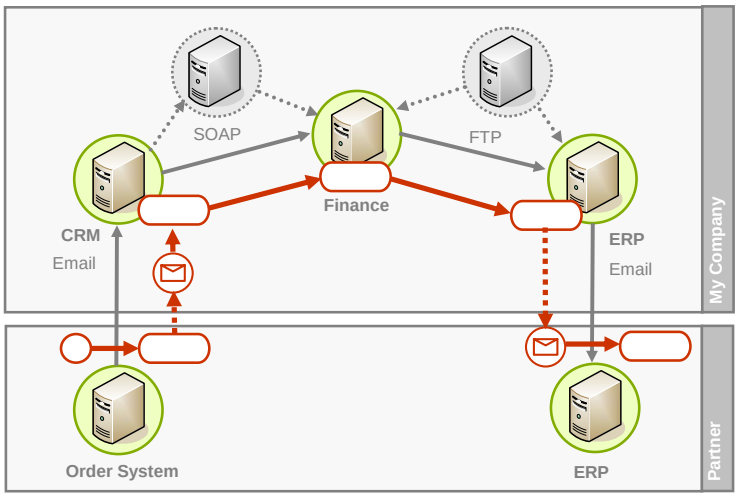


**Fig. 1.** An example of an accidental architecture that consists of tightly-coupled applications that are connected via unreliable point-to-point connections. The process and data transformation logic is encoded into the applications.

The EAI integration approach tries to integrate all kinds of applications using a centralized EAI broker. As we can observe in most enterprises today, this leads to so called islands of integration. They exist because at a certain point in time even the most ambitious and best-funded EAI integration project fails, because the heads of the departments refuse to give up control over their resources through integrating them into or moving them to a centralized infrastructure controlled by the enterprise. However, inside this islands of integration, most of the aforementioned point-to-point integration problems are already solved.

Thus, the resulting architecture is named accidental not only because it has been developed through a number of "accidents" but also because it is very accident-sensitive through the aforementioned characteristics.

### 2.3 What does the ESB promise?

The ESB promises to construct a SOA by iteratively integrating all kinds of isolated applications into a decentralized infrastructure called service bus. In general, ESB is based on ideas from EAI, in special message routing and transformation. But, because of the decentralized infrastructure, it does not force departments to integrate their applications into a centralized EAI broker and, therefore, to loose control. It rather allows departments to provide selective access to their business functionality and information, to enforce local policies and, therefore, to keep local autonomy.

Iterative integration means that the ESB does not follow an all-or-nothing approach. Because of the infrastructure that is not only decentralized but also

highly distributed and versatile, it rather allows to bring all kinds of applications step-by-step to the service bus. Therefore, the integration projects now have the goal to bring the business functionality implemented by different applications as reusable business services to the bus. These business services can then not only be used in the current integration project but also reused and composed in subsequent projects. The main difference compared to former EAI solutions is that the conducted integration projects now follow a long-term strategy, that is to bring all kinds of enterprise applications as business services to the service bus.

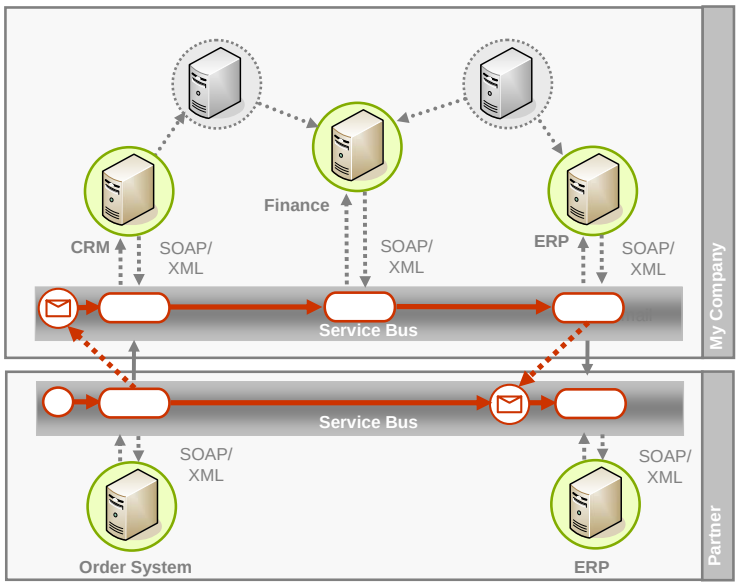


**Fig. 2.** An example ESB architecture in which all kinds of applications are provided as business services and connected via reliable, secure and manageable virtual channels. As a consequence, process orchestration and data transformation logic can be moved to the bus and process interactions can be performed in a controlled manner.

Technically, the main difference between the ESB and former EAI solutions is that it replaces all direct application connections through reliable, secure and manageable virtual channels. Through the introduction of these virtual channels the applications are also decoupled, which leads to loosely coupled interactions and interfaces. To allow a standardized message exchange between different business services, ESB also propagates the use of XML as data format and SOAP [26] as message exchange protocol.

As a consequence of this changes, process orchestration and data transformation logic can be moved from the applications to the service bus. Because of that, the ESB now can also perform process interactions (choreographies) be-

tween a company's processes and its business partner's processes in a controlled manner. Figure 2 illustrates the result of refactoring the accidental architecture from figure 1 to an ESB architecture.

## 3 The Nature of an ESB

Having clarified what ESB promises, I now explain how these promises are realized. I will therefore give an overview about the key components of the ESB architecture and discuss each component in detail.

### 3.1 The Key Components of the ESB

The key components of an ESB architecture are MOM, service container and management facility. Figure 3 illustrates these key components and their relationships.

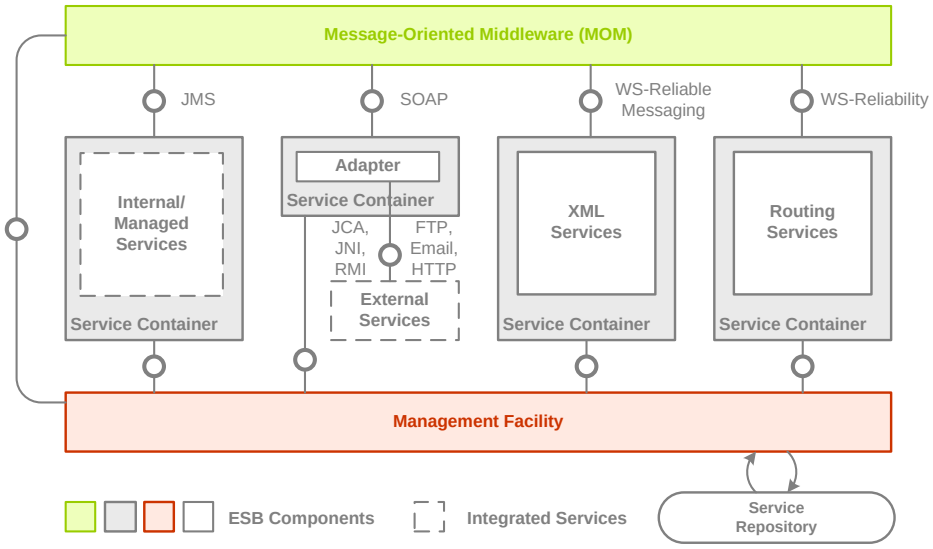


**Fig. 3.** FMC diagram of the ESB architecture that shows the relationship between MOM, service containers and management facility.

The MOM is basically a highly distributed network of message servers and is, therefore, also called the backbone of an ESB. It allows to establish reliable, secure and manageable virtual channels and to send messages over them.

A service container either manages an application internally or provides access to an external application via an appropriate adapter. Adapters provide

access to all kinds of applications. They allow for example to upload and download files, to send and receive emails or to invoke a remote method via RMI. In all these cases, the service container makes the business functionality implemented by the managed application available as business services. It also connects these business services to particular virtual channels and therefore allows them to send and receive messages over the MOM. Both, intelligent service containers and highly distributed MOM give the ESB its decentralized nature.

In an ESB architecture, a number of special services are available by default. Among these are routing and XML processing services. As the integrated services, they are managed by service containers and connected to certain virtual channels.

Software engineers can easily use, reuse and compose business services by establishing virtual channels and connecting the right business services to them. In order to do that, MOM and service containers need to be configured.

Therefore, ESB has a powerful management facility to which MOM and all service containers are connected. Because of that the management facility knows all business services and virtual channels and allows to configure and monitor them.

### 3.2 The MOM

The MOM is the most important component of an ESB and I explain it therefore first. In this section, I will answer the following questions: "What is the benefit of having a MOM?", "How does a MOM function?", "How are virtual channels established?" and, finally, "What are the main characteristics of a message?".

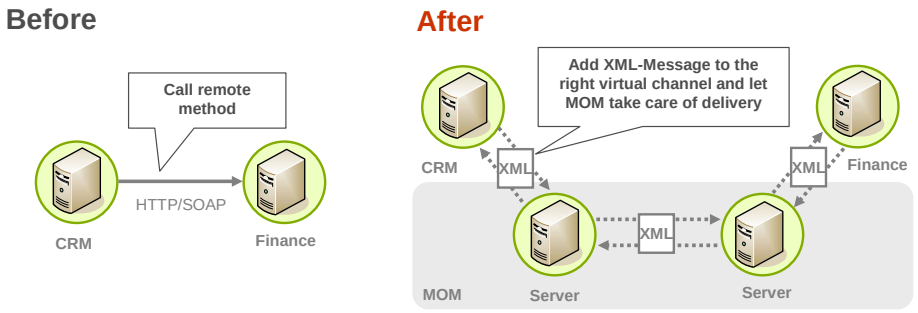


**Fig. 4.** The shift from synchronous remote calls to asynchronous message exchange.

**The Benefit of Having a MOM: Reliable, Asynchronous Message Exchange** As aforementioned, in an ESB, all direct communication channels between applications are replaced by virtual communication channels. As a result,

all synchronous remote calls are replaced by asynchronous message exchange. Because of that, all tightly-coupled point-to-point interactions are replaced by loosely-coupled indirect interactions. The MOM actually takes care of sending the messages via the setup virtual communication channels to the connected business services. Figure 4 illustrates the shift from synchronous remote calls to asynchronous message exchange and the resulting impact.

**Sending Messages over the MOM** The MOM that actually takes care of the message delivery consists of a network of message servers and a number of message clients. Figure 5 illustrates that on an example setup.

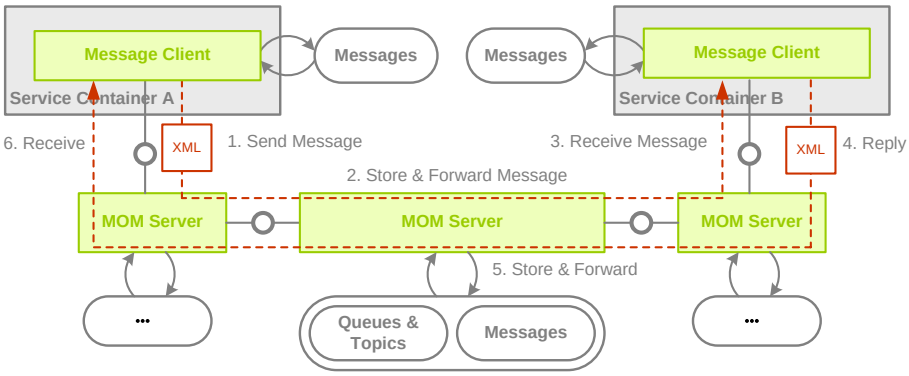


**Fig. 5.** FMC diagram illustrating how messages are sent over the MOM.

A message server basically manages various queues and topics and is able to store messages sent to those. An ESB often consists of multiple message servers that are connected to each other. The MOM routes the messages reliably through this network of message servers via a store and forward mechanism. This means, that each message server on the route stores the message, tries to send it to the next message server and deletes it only if the target server has acknowledged the reception. Using this mechanism, the MOM can guarantee the message delivery.

Each message client is connected to a message server and runs inside a service container. Because of that, it is able to send messages to and receive messages from this message server. However, the service container actually manages the message client and takes care of transforming the received messages into service invocations. Most message clients are also able to store messages temporarily.

There are different messaging standards and APIs, that can be used to send messages to and receive messages from a MOM. Using JMS [19] in conjunction with SOAP is very popular but only works in a Java environment. Therefore, upcoming standards such as WSRM [24] pose a promising approach for the future.

**Establishing Virtual Channels in a MOM** As aforementioned, a message server is able to manage topics and queues. They are either used to realize a point-to-point or a publish-subscribe messaging model. Figure 6 illustrates the use case and the technical realization of both models.

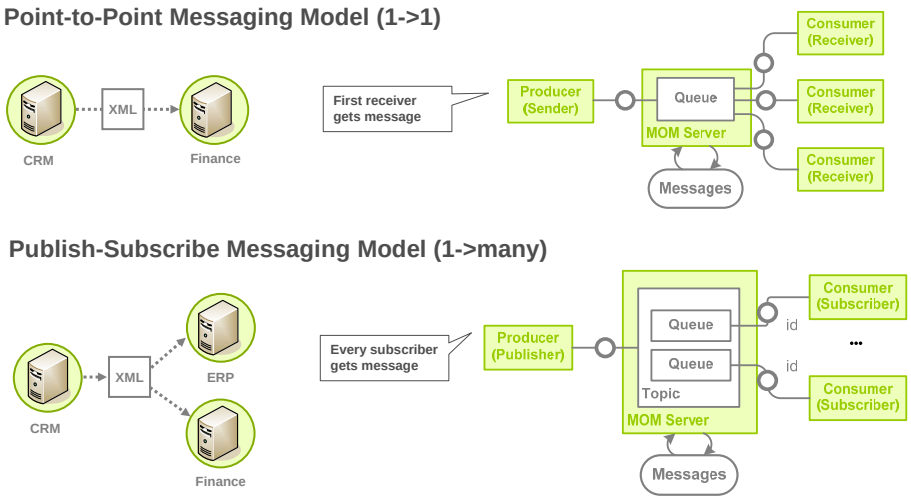


**Fig. 6.** Establishing virtual channels using either point-to-point or publish-subscribe messaging models.

One can use a queue to realize a point-to-point messaging model. Therefore, a message sender sends messages to a queue. The queue just exists virtually and is managed by a message server. This message server also stores the received messages temporarily. To receive messages, a message receiver can connect to a queue and fetch the oldest message. But, although multiple message receivers might be connected to this queue, the oldest message in the queue will only be delivered to that message receiver that fetches the message first. As you can see in figure 6 this messaging model can be used to establish a virtual channel between two applications.

One can use a topic to realize the publish-subscribe messaging model. Similar to the point-to-point messaging model, a message publisher publishes messages to a topic. Message subscribers can subscribe to that topic to receive the published messages. However, in this case, the message server manages the virtual topic and for each subscriber a private queue in which the messages are stored. Therefore, not only one but all subscribers receive the published message. As you can see in figure 6, this model can be used to establish a virtual channel between multiple applications.

Through the intelligent connection of queues and topics that are managed by different message servers, one can also establish virtual channels between

business services provided at different geographic locations. As aforementioned, the MOM will take care of the reliable message delivery.

**Messages: The Means to Transport Data** In an ESB, messages are the basic unit of transaction. Because they are sent instead of direct method invocation, they have to contain more information than the plain data to be transmitted.

Therefore, a message consists of a header, properties and a body. The header contains identification and routing information. The properties allow to pass application-specific values. Typical message properties are replyTo, correlationId and messageId attributes. The body, finally, contains the actual payload of the message.

The ESB is based upon a standardized message exchange. This means that messages are sent in a normalized format. Therefore, on a business service invocation, they might have to be transformed from the normalized format to the format required by the business service and vice versa.

Because the SOAP message exchange protocol has exactly the aforementioned characteristics and is standardized, it is used in most ESBs to send messages across the network. But SOAP is not mandatory and other, sometimes proprietary protocols, can also be used. However, using a proprietary protocol might lead to a vendor look-in and therefore to islands of integration, again.

The message payload often contains XML documents although this is not mandatory, too. But the advantage of using XML and the reason why it is used in most cases is that it allows to easily transform the contents and route messages based on the contents through the service bus.

### 3.3 The Service Container

In an ESB, the service container is the means to service-enable all kinds of applications. It is connected to topics and queues provided by the MOM and is able to transform messages into service invocations. It service-enables applications, that are either managed internally by the container or managed externally and adapted by the container, by providing the business functionality of these applications as loosely-coupled, coarse-grained business services. Therefore, a business service can encapsulate very different functionality, such as to upload or download a file from an FTP server, to send or receive an email from a mail server, to invoke a method on an EJB, to invoke a method on a simple Web service or to invoke a method on a SAP R/3 instance using a JCA [23] adapter.

Each business service is represented by an ESB endpoint, has a unique endpoint address which can be used to reference it, and is registered at the distributed management infrastructure. Because of that, they can be used to route messages to and compose business processes out of them. An ESB endpoint can be represented by a Web service but does not necessarily have to.

Service containers are not a unique feature of ESBs. They have been used for years in EAI solutions. They are also used for example in J2EE [22] to manage JSPs, Servlets and EJBs. Recently, new lightweight containers such as the ones

provided by the Hivemind [37] and Spring [36] project have become very popular. Each of these service containers can in general be used in an ESB, as long as it can be connected to the MOM and can be managed by the management facility.

**Connecting Services to the ESB** As we already know, the service container manages a message client, that allows to send and receive messages from certain queues and topics, and it manages or adapts an application. It also manages a number of ESB endpoints. These ESB endpoints are the mediators between message client and the application's business functionality. David Chappell describes in his book "Enterprise Service Bus" [1] a special kind of ESB endpoint that I want to explain here as well. Figure 7 illustrates this ESB endpoint approach.

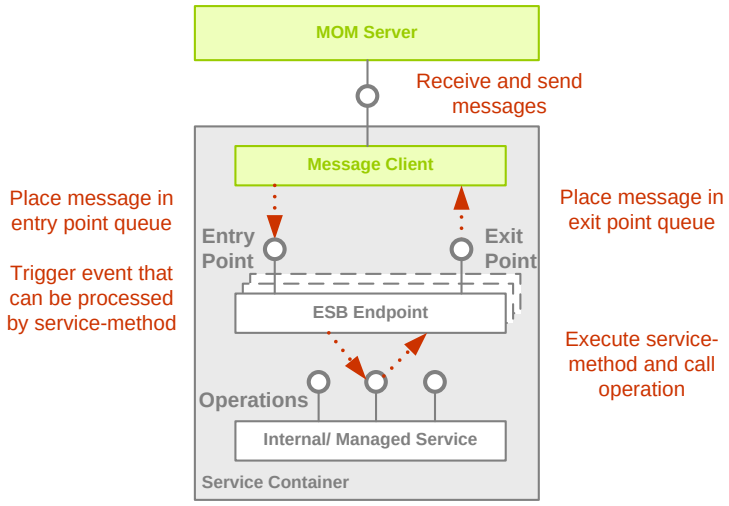


**Fig. 7.** Connecting serivces to the ESB using ESB endpoints that are managed by a service container.

An ESB endpoint is similar to a Servlet in J2EE. It has a standardized interface that consists of an entry point and an exit point. The service container places all received messages in the queue of the entry point and messages that shall be sent in the queue of the exit point. Each ESB endpoint has a service method. The service method is called each time an arriving message triggers an event that has to be processed. Calling the service method, the only input parameter is an ESB context that allows to access messages in the entry point queue and place messages in the exit point queue. Finally, the service method contains the code that handles the received message and might send new messages or error message to the MOM. However, using the ESB context arbitrary messages can be send to the MOM.

The code of the service method can for example transform the received XML message into a Java object and call a specific method on the managed Java application. As aforementioned all kinds of integration tasks can be achieved using this approach. There are also a variety of default ESB endpoint implementations available that allow the integration of all kinds of applications by simply setting up some configuration parameters.

**Possible Capabilities of the Service Container** The core functionality of each service container is that it manages a message client to send and receive messages, that it has a management interface that allows to configure, manage and control the container, that it manages a number of configured ESB endpoints and that it has a simple service invocation framework that allows to call the service method on these ESB endpoints. However, having a service container, that manages the translation of pure messages send over the MOM into service invocations, allows to add almost arbitrary functionality in between, as long as it is manageable via the management interface. Figure 8 illustrates the possible capabilities of a fully-blown service container.

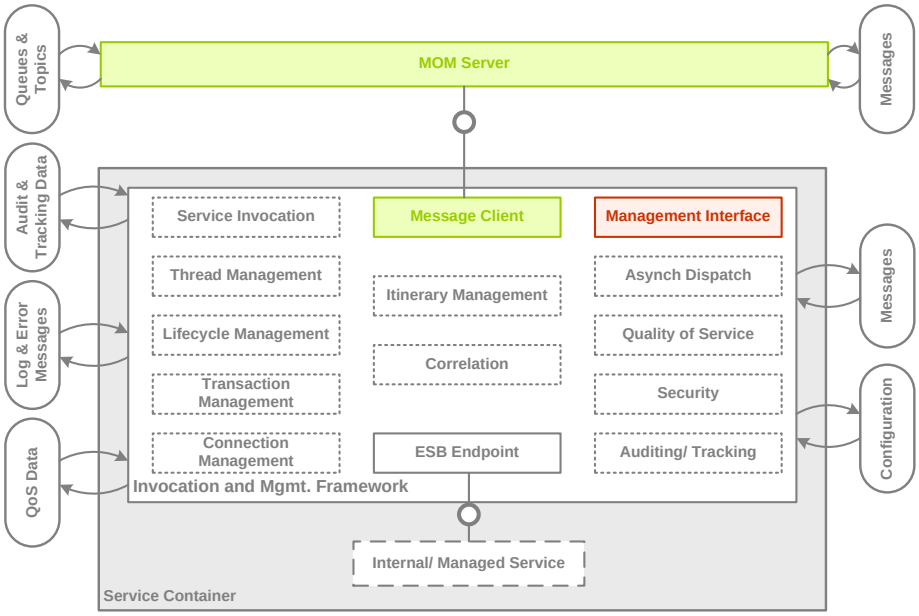


**Fig. 8.** FMC diagram that shows the possible capabilities of a service container.

Each service container can additionally provide functionality for auditing, tracking, logging and error handling. Besides this functionality, one can also add QoS functionality that allows to measure all kinds of service invocation relevant

data, such as the average service execution time, the throughput of the service or the average usage of the service.

Additional functionality might also be responsible for handling the security configuration that is required for accessing the MOM and the adapted, external applications.

For internally managed applications, the service container might also manage a thread or object pool in order to allow a faster request processing.

Itinerary management basically allows to handle itineraries, as it will be explained in section "Itinerary-Based Routing". Correlation handling means that the service container is able to correlate request and a corresponding response messages using certain correlation ids.

Besides the described functionality a service container might provided functionality for lifecycle management, transaction management, connection management and much more.

There are many organization that try to standardize the capabilities of a service container. One standard is JBI [21], which is the result of a Java community process and widely accepted in the Java world. WSRF [17] is another emerging standard, that is based on Web services standards.

### 3.4 The Management Facility

The ESB is based upon a highly distributed and decentralized infrastructure that consists of many service containers, that provide the business functionality of the managed applications as business services, and a MOM to which all service containers are connected and that connects these business services by establishing virtual channels between them. Thus, the service containers and the message servers of the MOM need to be configured, managed and monitored.

Because of the variety of managed applications, ranging from simple EJBs that are deployed via an deployment archive to transformation engines that require XSLT [29] scripts to BPEL [16] engines that require process definitions, and possibly different message servers, very different requirements concerning configuration, deployment, management, and monitoring have to be satisfied.

The basic idea of an ESB is to have a decentralized infrastructure but to manage it centrally. Each ESB, therefore, has a powerful and versatile management facility. This management facility basically consists of a central repository, a network of management servers, management interfaces at message servers and service containers, and different configuration, management and monitoring tools. Figure 9 illustrates the relationship between the aforementioned components.

**The Central Repository: The Means to Store all Kinds of Artifacts** In the central repository, all kinds of ESB related artifacts are stored. Besides the ESB endpoint configuration for the service containers and the topic and queue configuration for the message servers it also contains program code, deployment descriptors, deployment archives and XSLT scripts. The central repository also

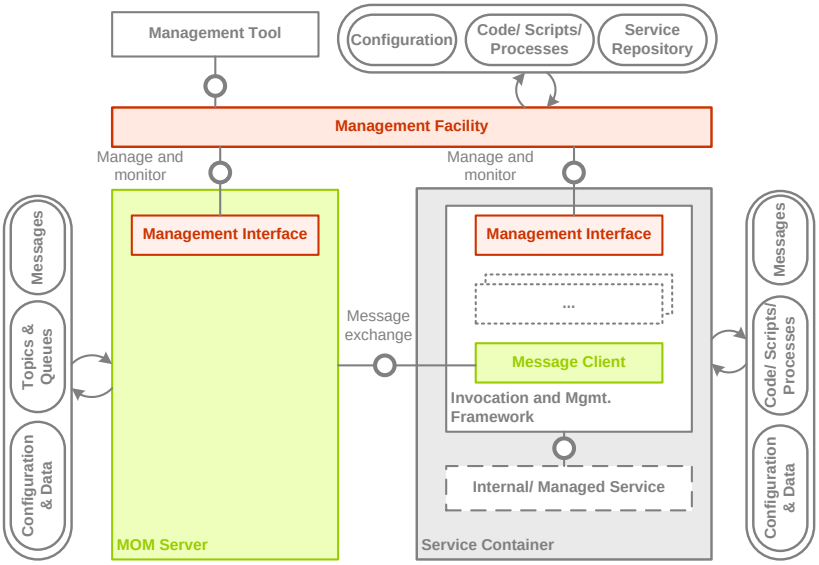


**Fig. 9.** FMC diagram illustrating the key components of the management facility of an ESB.

contains a list of available business services and their ESB endpoints, BPEL process definitions and message routing configurations. Because the management facility monitors all kinds of components, it also contains different monitoring data.

**The Network of Management Servers: Managing the Decentralized Infrastructure** The management facility is built upon a network of management servers. These servers are connected to the central repository. In case of a globally distributed ESB, the management servers can also replicate the data to different physically separated repositories.

Each message server or service container is connected to one of the management servers. They therefore cannot only read data from and write data to the central repository but also to all connected components. These components can store certain data locally. Because of having a central repository on the one hand and storing the appropriate data for each component locally, the management facility is very robust.

A management server basically configures the connected components, deploys files on them, monitors them and manages them in general. Configuration means, that it configures topics and queues of the message servers and the ESB endpoints of the service containers. It also stores the ESB endpoint references along with the business service description in the central repository. However, other aspects

like logging, error handling, auditing, QoS and security can also be configured. What can be configured basically depends on the capabilities of the service container or message server.

Deployment means, that the management server can upload all kinds of files to a service container. To deploy an EJB in a service container it uploads for example a Jar file, to configure a transformation engine it uploads certain XSLT scripts and to configure a BPEL engine it uploads specific BPEL process definitions.

Besides the configuration and deployment aspect, a management server can also monitor the connected component and collect all kinds of management data. Among these data are for example life-cycle, log, error, auditing or QoS data.

Finally, a management server can also manage the life-cycle of the connected components. It therefore can, for example, start, stop and restart the connected components.

**The Management Interface: Providing Access to all Kinds of Components** Each message server and service container has a management interface that basically provides configuration, deployment, monitoring and lifecycle management functionality.

The management interface can be based, for example, on the older SNMP [18], on the popular JMX [20] or on the latest WSRF [17] management standard.

**The Management Tools: Configuring, Monitoring and Managing Components** The management tools allow human beings to access the data that is stored in the central repository, and the message servers and service containers that are connected to the management infrastructure.

Using a management tool, software engineers or business process specialists can for example view all available business services that are stored in the central repository, including their description and ESB endpoint reference. They can use these services to compose them, in order to create and automate business process. Because business processes that are managed by a BPEL engine can be accessed like any other business service, one can also define process interactions resp. choreographies.

Having these management tools, one can also monitor the components to see which components are available and which ones are down. Using the lifecycle management functionality provided by the management interface one can also start, restart and stop these components. The management tools also allow to monitor and manually handle errors that occured in the message or process flow. Therefore, they allow to access the service container or message server where the processing error occured and to edit for example the XML content of the message by hand.

The management tools also allow to view all kinds of monitoring data, for example QoS, error, log and auditing data, to create statistics based on that data and to visualize them as graphics.

# 4 Special Facilities of the ESB

Until now, I have only described the basic functionality of an ESB. However, the goal of an ESB is to integrate all kinds of isolated applications into a decentralized infrastructure to provide the business functionality as reusable business services, to create, automate and integrate business processes using them, and to manage and monitor the created business processes. Because messages sometimes have to be pre-processed before and post-processed after service invocation, and business services and business processes have to be enacted somehow, the ESB provides special routing and XML processing facilities. I will explain them in the following section in detail.

## 4.1 Routing Facilities

Through the usage of a MOM, an application's functionality is no longer executed based on a direct, synchronous method invocation but on an indirect, asynchronous message exchange. This message exchange is always conducted between a business service and the service bus. So, somehow the service bus needs to know how to route the messages through the bus.

Therefore, an ESB basically provides three mechanisms to route messages through the bus thereby invoking multiple business services: itinerary-based routing, service orchestration using BPEL and content-based routing. This mechanisms allow not only to manage the business processes but also to monitor them.

**Itinerary-Based Routing** Itinerary-based routing is often used to manage short-living, transient process fragments. Gartner Inc. calls this process fragments *microflows*. A microflow consists of a sequence of logical steps. Each logical step refers to a business service. Thus, to enact a microflow, a message is sent through the service bus in such a way that all business services are invoked. Therefore, one must think of the service bus as a highly distributed routing network that is build up by a variety of message servers and service containers.

In order to route a message through the bus, each message contains an itinerary. The itinerary consists of a list of ESB endpoints that have to be visited and the information about already visited ESB endpoints. The message also contains the current processing state as message payload. Because the itinerary and the process state is carried by the message as it travels across the bus, each service container is able to evaluate the itinerary and to decide in which virtual channel the message has to be placed, to send it to the next ESB endpoint in the list. Figure 10 illustrates this approach.

The advantage of using the decentralized routing network is that different parts of the network can operate independently of one another without relying on any centralized routing engine. Because of the decentralized nature of this approach, there is no single point of failure or performance bottleneck.

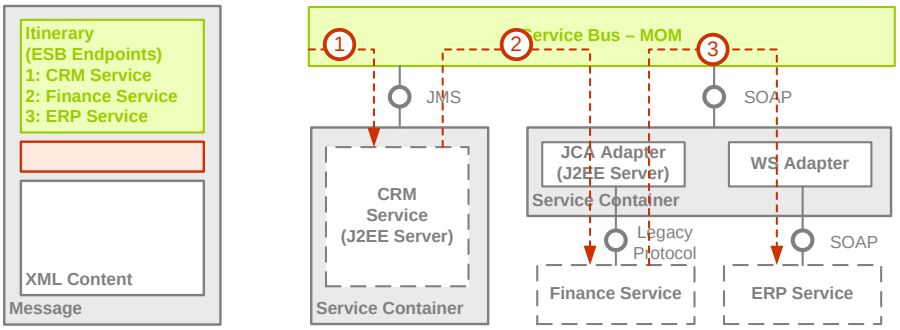


**Fig. 10.** Itinerary-based routing in an ESB.

**Service Orchestration using BPEL** Service orchestration using BPEL is used to manage long-running business processes that might run for months or years. A BPEL process definition consists of a number of logical steps that are connected to each other by conditional or unconditional links and can be executed in sequence or in parallel. A BPEL process definition also allows to define time-based, condition-based and event-based triggers. As in the itinerary-based routing, each logical step refers to an ESB endpoint.

A service orchestration or BPEL engine is used to enact BPEL processes based on the process definitions. The BPEL engine is provided by the ESB as a special service via an ESB endpoint an can therefore be accessed like any other service. Depending on the setup, an ESB might contain multiple BPEL engines in different geographic locations that manage different BPEL processes.

During enactment, the BPEL engine simply sends asynchronous messages to and receives asynchronous message from the MOM. Depending on the kind of logical step, it can thereby invoke a business service or interact with a business process managed by another BPEL engine. The procedure of invoking a business service follows the find-bind-invoke mechanism. This means, that the BPEL engine finds the required business service by resolving the defined ESB endpoint, binds to it, and, finally, invokes it by sending a message. To emulate a synchronous service invocation, the BPEL engine is also able to correlate the send request with a reply message of the invoked service.

Figure 11 illustrates an example of a service orchestration. The part above shows the message exchange between BPEL engine and MOM as a BPMN diagram. The part below shows how the message exchange between BPEL engine service and business services actually functions as an FMC block diagram.

As you might have noticed already, the BPEL engine not only manages the process definitions but also the state of the currently enacted processes. Therefore, service orchestration can be used to handle more complex situations than with a simple itinerary. Such complex situations occur, for example, when a stateful conversation between two business processes is carried out over a long

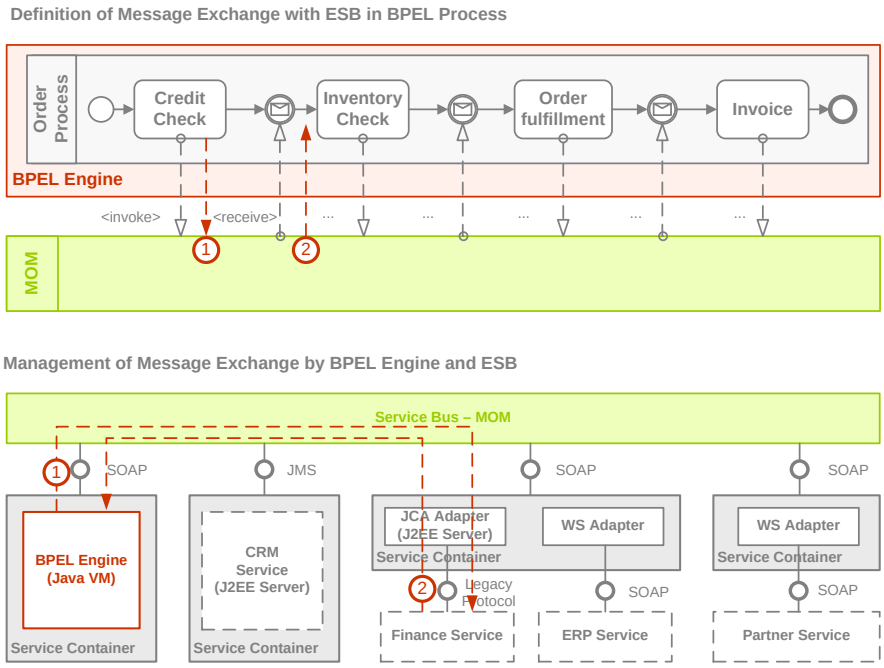


**Fig. 11.** Service orchestration using BPEL.

duration with pauses and resumes that are separated by time and triggered by external events.

The disadvantage of the service orchestration through a centralized BPEL engine is that it represents a possible single point of failure and a performance bottleneck. The advantage of this approach is obviously, that through the central process management, failure and recovery can be handled and processes can also be suspended for a certain time.

Although I talked about service orchestration using BPEL in this section, you can orchestrate services without using BPEL as well. However, although the BPEL standard has many flaws, it is adopted as the process definition standard by the industry. Therefore, to avoid a vendor lock-in by using a proprietary language you should use BPEL.

**Content-Based Routing** Content-based routing (CBR) is based on the fact that XML processing services with different capabilities are plugged into the bus. They basically allow to validate, enrich, transform, route and operate XML messages. Combinations of these services allow to form lightweight processes with the sole purpose to process messages.

Plugging such a lightweight process as CBR service into the message flow between a message producer and a message consumer (which might be for ex-

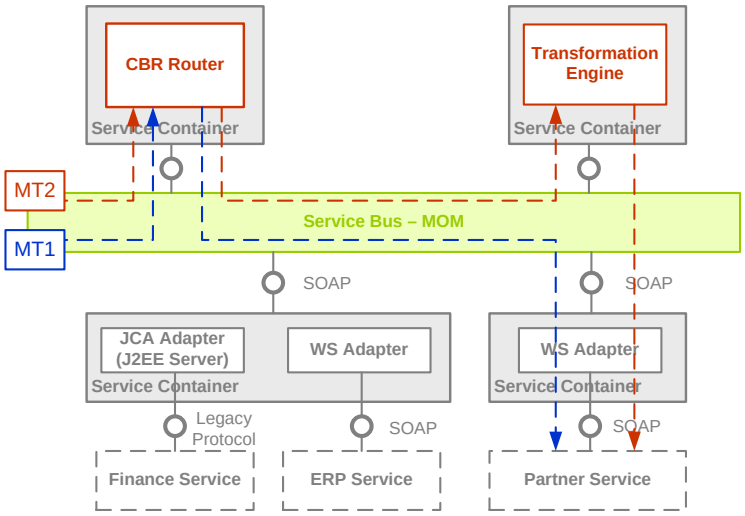


**Fig. 12.** Content-based routing in an ESB.

ample two business services) allows to handle all kinds of complex integration tasks, for example before and after a service invocation.

Figure 12 shows a content-based routing example. As you can see, a CBR router and a transformation engine are plugged into the bus between a message producing service and the partner service. The sole purpose of the CBR router is to apply an XPath [27] expression to determine whether the message conforms to message format M1 and to sent it to the transformation engine if the message format is M2. The transformation engine then basically transforms message format M2 to M1 by transforming for example a 5 digit postal code to a 9 digit one.

### 4.2 XML Processing Facilities

Because an ESB is used to integrate all kinds of applications, and for really integrating an application it might require more than one simple step, the ESB provides a wide range of XML processing facilities, that can be plugged together as described in the content-based routing section to handle all kinds of complex integration tasks.

These XML processing facilities allow, among others, to validate, transform, and persist messages. They are either provided by the service containers or by special XML services that are plugged into the bus.

Message validation means that validation services are plugged into the bus that are capable of checking whether a message conforms to a certain message or data format. Therefore, it either checks the XML payload for the existence of certain attributes and tags or evaluates the contents using configured validation

rules. Validation services have often some routing intelligence and transformation capabilities that allow them to modify the processed message or the routing information (the itinerary) of the processed message based on the validation result. In order to validate messages, validation services use XML standards, such as XPath or XQuery [28].

Message transformation means that transformation services in the bus or transformation functionality implemented in the service container is used to change, extract, enrich or aggregate the XML payload of the processed messages. In order to do that, transformation facilities use XML standards, such as XSLT, XPath or XQuery.

Message persistence means that special services are plugged into the bus that are connected to XML or relational databases and are able of storing XML messages or their payload.

As mentioned above, XML processing services are mostly used in content-based routing scenarios. However, because these services are accessible via an ESB endpoint like any other service in the ESB, they can easily be used in microflows and BPEL processes, as well.

## 5 Conclusion

In this paper, I gave an introduction to the ESB. Therefore, I described what the basic promises of ESB and the main differences to former EAI solutions are. I explained the key components of an ESB, which are MOM, service container and management facility, in detail. I also described the special facilities, which are routing and XML processing facilities, that actually make up an ESB.

As a conclusion of my work, I can say that ESB combines best practices from EAI of the last years, reuses and integrates components that have been on the market for years, and makes it more manageable. It combines best practices from EAI because it is based on concepts from MOM, event-driven architecture (EDA) and SOA. It reuses components, such as messaging systems, J2EE servers, integration adapters from centralized EAI solutions, business process management engines and XML processing services, and integrates them to provide added-value. Finally, it makes the integrated components more manageable and therefore more valuable by providing a powerful management facility and integrating them into it.

### 5.1 What is (an) ESB?

Having clarified the advantages and disadvantages of an ESB, let us finally answer the question: "What is (an) ESB?".

**A "Way of Doing Things"?** Yes, the ESB is definitively a "way of doing things". It is an incremental approach of constructing a SOA by connecting all kinds of applications to a enterprise-wide distributed infrastructure.

**An Architecture?** Yes, the ESB is an architectural style in which applications are service-enabled through service containers and connected to a MOM based service bus, that is not only capable of routing messages but also of transforming them. This architectural style allows to iteratively construct a SOA, to create, automate and integrate business processes based on the provided business services, and to easily manage and monitor these business processes.

**A new Type of Product?** Yes, somehow. There are many companies that sell ESB infrastructure products allowing enterprises to build up an ESB. These products are often composed out of existing components, such as MOMs, J2EE servers and EAI integration adapters, and provided in a manageable manner.

Software companies, such as IBM [30], Sonic Software [31], Seebeyond [32] and Cape Clear [33] are very active participants in this market. They are fighting for market shares by selling there own ESB infrastructure products and offering consulting services to help enterprises in realizing their ESB.

There are also a number of open source projects, such as Open ESB [34] sponsored by Sun and Mule [35] sponsored by Codehaus that try to provide enterprises with free ESB infrastructure implementations.

## References


1. Chappell, D. A.: Enterprise Service Bus. O'Reilly Media Inc., 2004.
2. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Pearson Education, 2004.
3. Fowler, M.. Patterns of Enterprise Application Architecture. Addison Wesley, 2002.
4. Pulier, E., Taylor, H.: Understanding Enterprise SOA, Manning, 2006.
5. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall, 2004.
6. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer-Verlag, 2004.
7. Tabeling, P., Groene, B., Knoepfel, A.: Fundamental Modeling Concepts - Effective Communication of IT Systems. John Wiley & Sons, Ltd., 2006.
8. PolarLake: Understanding the ESB. http://www.polarlake.com/en/assets/whitepapers/esb.pdf.
9. Sun Microsystems: Service Oriented Business Integration. http://java.sun.com/integration/.
10. Papazoglou, M.P., Georgakopoulos, D.: Service Oriented Computing. Communications of the ACM, 46(10), 2003.
11. Chappel, D.A.: Using the ESB Service Container. http://www.onjava.com/pub/a/onjava/excerpt/esb_ch6/index.html, 2004.
12. Keen, M. et al.: Implementing an SOA using an Enterprise Service Bus. http://www.redbooks.ibm.com/redpieces/pdfs/sg246346.pdf, 2004.
13. Keen, M. et al.: SOA with an Enterprise Service Bus in WebSpehere. http://www.redbooks.ibm.com/redbooks/pdfs/sg246494.pdf, 2005.
14. Business Process Management Initiative: BPMN: Business Process Modelling Notation 1.0. http://www.bpmn.org/Documents/BPMN V1-0 May 3 2004.pdf.

15. FMC Consortium: FMC: Fundamental Modelling Concepts.
http://www.f-m-c.org.
16. BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems: Business Process Execution Language for Web Services 1.1 (BPEL4WS).
http://www- 128.ibm.com/developerworks/library/specification/ws-bpel/.
17. The Web Services Resource Framework.
http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrfpaper.html
18. The SNMP Protocol.
http://www.snmp.com/protocol/
19. Sun Microsystems: Java Message Service (JMS) API.
http://java.sun.com/products/jms/.
20. Sun Microsystems: Java Management Extension (JMX).
http://java.sun.com/products/JavaManagement/
21. Sun Microsystems: Java Business Integration (JBI).
http://www.jcp.org/en/jsr/detail?id=208.
22. Sun Microsystems: Java Enterprise Environment (Java EE).
http://java.sun.com/javaee/.
23. Sun Microsystems: Java EE Connector Architecture (JCA).
http://java.sun.com/j2ee/connector/.
24. Oasis: Web Service Reliable Messaging (WSRM)
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm
25. W3C: Web Services Architecture (WSA).
http://www.w3.org/TR/ws-arch/, 2004.
26. W3C: SOAP specification.
http://www.w3.org/TR/soap/.
27. W3C: XML Path Language (XPath).
http://www.w3.org/TR/xpath.
28. W3C: XML Query Language (XQuery).
http://www.w3.org/TR/xquery/.
29. W3C: XML Transformations (XSLT).
http://www.w3.org/TR/xslt.
30. IBM Websphere ESB product page.
http://www-306.ibm.com/software/integration/wsesb/.
31. Sonic Software ESB product page.
http://www.sonicsoftware.com/products/sonic_esb/index.ssp.
32. Seebeyond eInsight ESB product page.
http://www.seebeyond.com/software/einsightenterprise.asp.
33. Cape Clear ESB product page.
http://www.capeclear.com/products/cc6.shtml.
34. Open ESB project page.
https://open-esb.dev.java.net/.
35. Mule ESB project page.
http://mule.codehaus.org/.
36. The Spring Framework.
www.springframework.org/
37. The Hivemind Framework.
http://jakarta.apache.org/hivemind/index.html