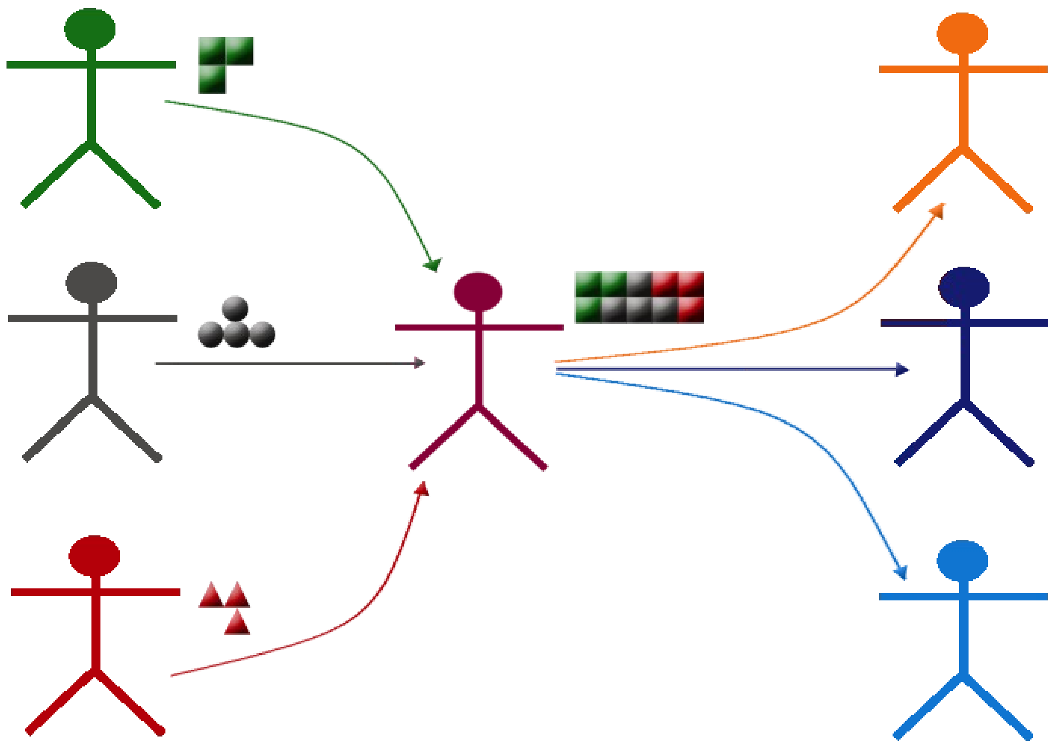
Towards Distributed Information Access

Possibilities and Implementation



Victor de Graaff

Towards Distributed Information Access

Alternatives and Implementation

Master thesis

Author: Victor de Graaff
University: University of Twente
Master: Computer Science
Track: Software Engineering
Internal Supervisors: Dr. Luís Ferreira Pires
Dr. ir. Marten van Sinderen
External Supervisors: ing. Gerke Stam, TSi Solutions

University of Twente
Enschede - The Netherlands

Preface

This thesis describes the results of a Master of Science assignment at the Software Engineering group at the University of Twente. This assignment has been carried out from February to November 2009 at TSi Solutions in Enschede, The Netherlands.

I would like to thank all the people who gave me support while writing this thesis. In the first place these people are my girlfriend Susanne Jeschke and my daughter Melissa. They have supported me through the last piece of my Bachelor and my entire Master course by taking my mind off work every night and weekend, as far as my deadlines allowed them to. They have given me the chance to spend countless hours on my work, my classes, and later my thesis, while they went to all the activities and appointments a young child has. I can truly say that I could never have finished my studies without their support, understanding and patience.

Another great influence has been the supervision of Luís Ferreira Pires, Marten van Sinderen and Gerke Stam. Luís managed to keep the balance in my work between quality and steady progress. On top of that, Luís taught me a lot on writing objective texts by rephrasing or pointing out sentences which were subjective or too popular. I owe him a red pen. Marten has helped me a lot to improve the first impression of the report, by pointing out missing balances in figures, texts and chapter structure. Gerke has been my big motivator for my thesis. His almost daily presence by my desk, at least with the coffee can, has forced me to not drift off with my attention to my work outside my thesis. Gerke also has a great ability to crawl inside the skin of someone who reads a text for the first time.

Although my Dutch family has been at some physical distance during my Master course, their emotional support has been of great value. My friends from the University of Delft, who all just graduated or are about to, have provided me with motivating competition.

Enschede, November 2009

Victor de Graaff

Brief Content

Preface

Brief Content

Extended Table of Contents

List of Figures

List of Tables

List of Listings

1. Introduction

2. Requirements

3. Comparison of Integration Technologies

4. Application Integration Architectures

5. Choosing an Implementation

6. Proof of Concept

7. Conclusion

References

Appendices

Extended Table of Contents

Preface....v
Brief Content....vii
1. Introduction....1
    1.1 Motivation....1
    1.2 Objectives....3
    1.3 Approach....4
    1.4 Report Structure....5
2. Requirements....6
    2.1 Approach....6
    2.2 Stakeholders....7
    2.3 Use cases....8
    2.4 Functional requirements....9
    2.5 Non-functional requirements....10
    2.6 Weighting Factors....11
3. Comparison of Integration Technologies....14
    3.1 Approach....14
    3.2 Point-to-Point Integration....15
    3.3 Hub-and-Spoke Integration....15
    3.4 Enterprise Message Bus Integration....17
    3.5 Enterprise Service Bus Integration....19
    3.6 Conclusion....21
4. Application Integration Architectures....22
    4.1 Hub-and-Spoke Architecture....22
    4.2 Enterprise Message Bus Architectures....23
        4.2.1 Bus combined with a load balancer....23
        4.2.2 Two buses....24
        4.2.3 One bus....24
    4.3 Enterprise Service Bus Architectures....25
        4.3.1 Bus combined with a load balancer....25
        4.3.2 Two buses....26
        4.3.3 One bus....26
    4.4 Comparison....27
5. Choosing an Implementation....31
    5.1 Approach....31
    5.2 Mule....31
        5.2.1 Tool Support....31
        5.2.2 Components....32
        5.2.3 Hello World example....32
        5.2.4 Requirements compliance....33
    5.3 Apache ServiceMix....33
        5.3.1 Tool Support....34
        5.3.2 Components....34
        5.3.3 Hello World example....34
        5.3.4 Requirements compliance....35

5.4 OpenESB....35
5.4.1 Tool Support....35
5.4.2 Components....36
5.4.3 Hello World example....36
5.4.4 Requirements compliance....37
5.5 Comparison....38
5.5.1 Comparison by Rademakers....38
5.5.2 Comparison by Biberger....39
5.5.3 Comparison based on information broker requirements....39
6. Proof of Concept....41
6.1 Problem situation....41
6.2 Basics of Mule....42
6.3 Configuring Mule for an information broker....44
6.4 Testing environment....48
6.4.1 Distributed Travel Search....48
6.4.2 External web service....48
6.5 Conclusion....50
7. Final Remarks....52
7.1 Conclusions....52
7.2 Future research....52
References....54
Appendices....58
Appendix A. Requirements Questionnaire....58
Appendix B. Hello World Request and Response....61
Appendix C. WSDL document for ESB Comparison....62
Appendix D. Load test for the Mule configuration....63

List of Figures

Figure 1: Information broker with own information storage....1
Figure 2: Interaction sequence of a successful purchase without distribution....2
Figure 3: Integration architecture without database....2
Figure 4: Interaction sequence of a successful purchase with distribution....3
Figure 5: Use case diagram information broker....8
Figure 6: Situation before integration....14
Figure 7: Point-to-point integration....15
Figure 8: Hub-and-spoke with new central server....16
Figure 9: Hub-and-spoke with two central servers....16
Figure 10: Enterprise Message Bus....17
Figure 11: Internal structure of an EMB [7]....18
Figure 12: Publish-and-subscribe mechanism....18
Figure 13: Single recipient....19
Figure 14: Enterprise Service Bus....20
Figure 15: Closer look at ESB [7]....20
Figure 16: Hub-and-spoke model for the integration broker....22
Figure 17: Enterprise Message Bus for communication with producers only....23
Figure 18: Two EMBs for separate communication with producers and consumers....24
Figure 19: One EMB for all communication....24
Figure 20: Enterprise Service Bus for communication with producers only....25
Figure 21: Two ESBs for separate communication with producers and consumers....26
Figure 22: One ESB for all communication....27
Figure 23: Interaction sequence of non-distributed TravelSearch....41
Figure 24: Outline of a Mule service [12]....42
Figure 25: Mule sequence diagram [12]....43
Figure 26: Mapping of sequence diagram onto service description....44
Figure 27: Main service in the Mule configuration (dtsService)....45
Figure 28: Specific provider configuration (externalServiceService)....46
Figure 29: Load test results....51

List of Tables

Table 1: Weighting factors for requirements....13
Table 2: Requirements compliance of the architectures 30
Table 3: Open source ESB comparison from [36]....38
Table 4: Open source ESB comparison from [4]. 39
Table 5: Open source ESB comparison based on information broker requirements....39

List of Listings

Listing 1: Java Class for Hello World examples....32
Listing 2: Mule Configuration for Hello World example....33
Listing 3: Bean definition for Apache ServiceMix configuration....34
Listing 4: BPEL process for Hello World with OpenESB 37
Listing 5: The Mule configuration for our prototype....48
Listing 6: Request transformations....49
Listing 7: Response transformations 50

1. Introduction

This chapter presents an overview of this research, which is carried out as the final project for my Master degree at the University of Twente. This chapter is further structured as follows: Section 1.1 contains the motivation for this research, Section 1.2 describes its objectives, Section 1.3 presents the adopted approach, and finally, Section 1.4 introduces the structure of the rest of this report.

1.1 Motivation

Nowadays many goods and services (e.g. books or trips, respectively) are being offered on-line by producers or resellers. In [40], a claim is made that in 2003, European consumers spent over $14 billions on-line on traveling, an increase of 44% over the past year. Due to the huge amount of offers with diverse characteristics, *information brokers* have appeared. The role of these information brokers is to retrieve information about services and products via the Internet from multiple vendor catalogs and databases [14]. They form a central access point for a specific type of offers, make them comparable, and provide their clients with one structured way to access them.

The options for an information brokers' technical architecture can be categorized in two groups. In the first category the offered information is stored in a centralized place, such as a database, maintained by the information broker, as depicted in Figure 1, where arrows indicate requests over a network connection. The requests on the left-hand side take place at a regular time interval to update the information in the database, independent of the requests on the right-hand side.

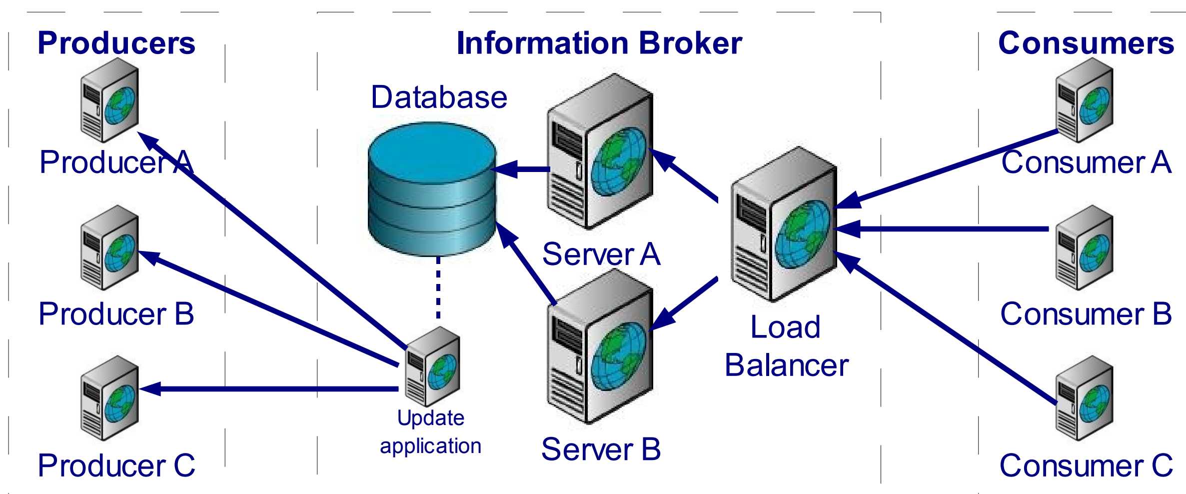


Figure 1: Information broker with own information storage

This category's main benefit of fast access to this information comes with the drawback that a *single point of failure* (SPOF) is created. Another challenge in this type of architecture is to keep the information in the database up-to-date, since other systems, for example systems of the producer, are simultaneously altering the original data.

A typical information broker also offers services to purchase the goods or services (from here on called *products*). An interaction sequence of such a communication for an information broker with his own information storage can be found in Figure 2. As seen in this figure, no connection needs

to be made to the producer during the retrieval of product information. The data exchange to populate the database takes place parallel to this process.

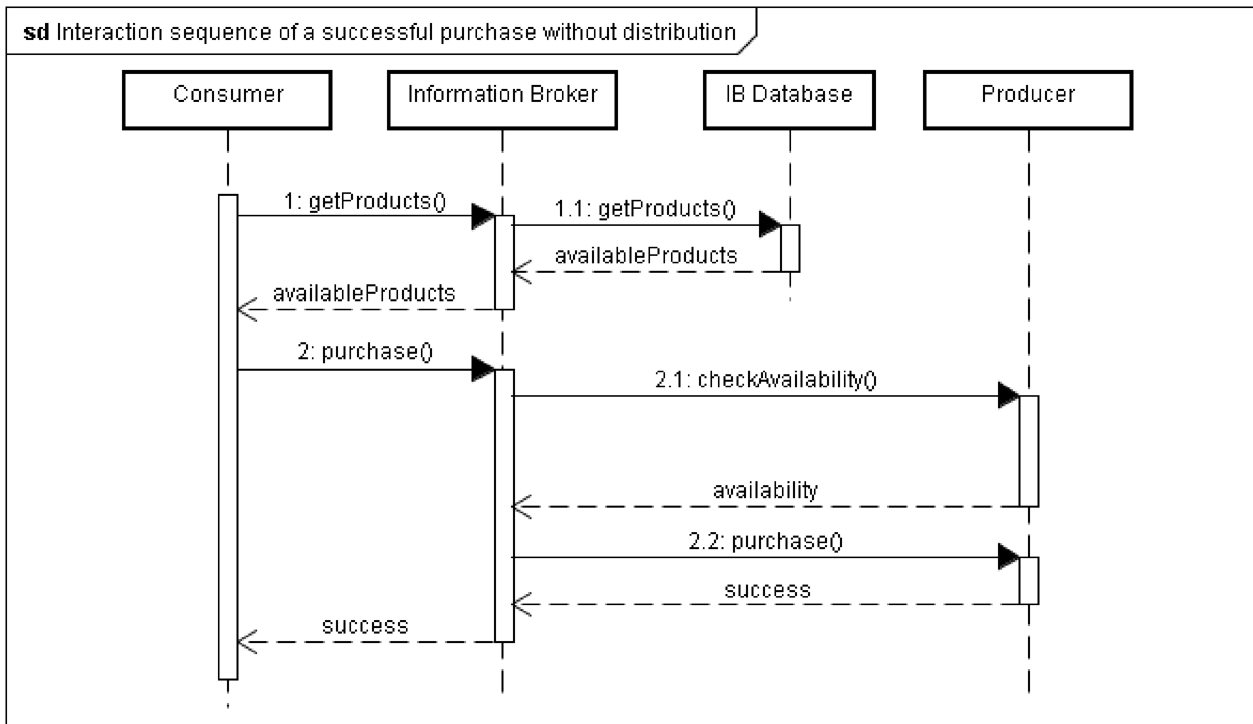


Figure 2: Interaction sequence of a successful purchase without distribution

In the second category of architectures, information is accessed at their source on real-time, in a distributed way, as can be seen in Figure 3. The arrows on the left-hand side indicate requests which are triggered for each request to the corresponding server on the right-hand side. This category allows the information broker to provide its *consumers* with up-to-date information. The drawback, however, is the increased access time.

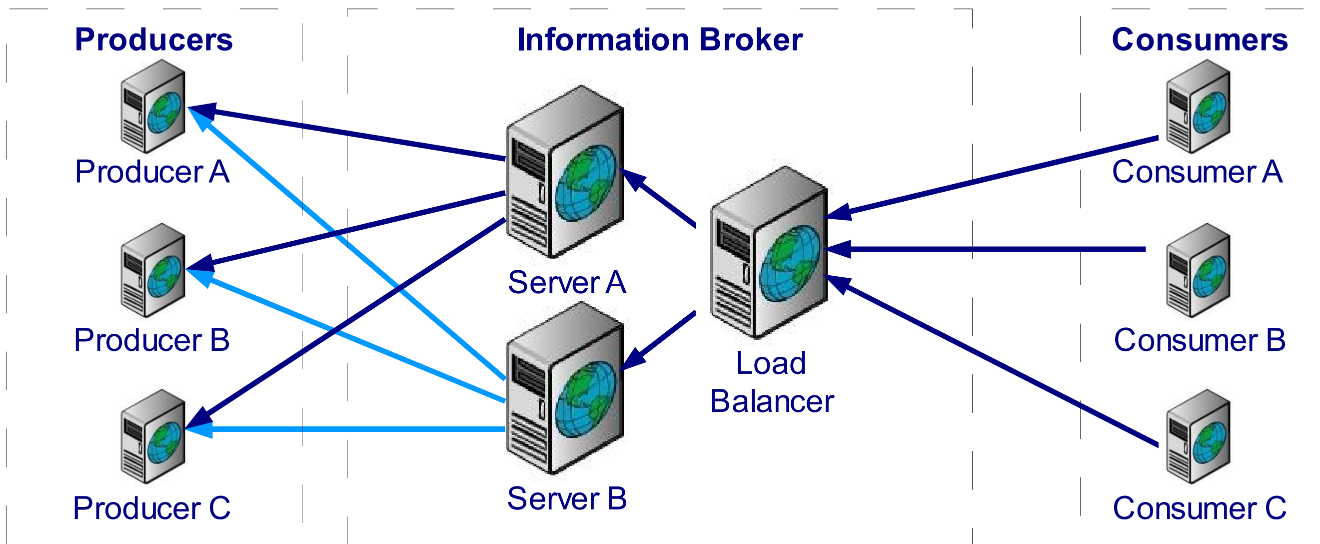


Figure 3: Integration architecture without database

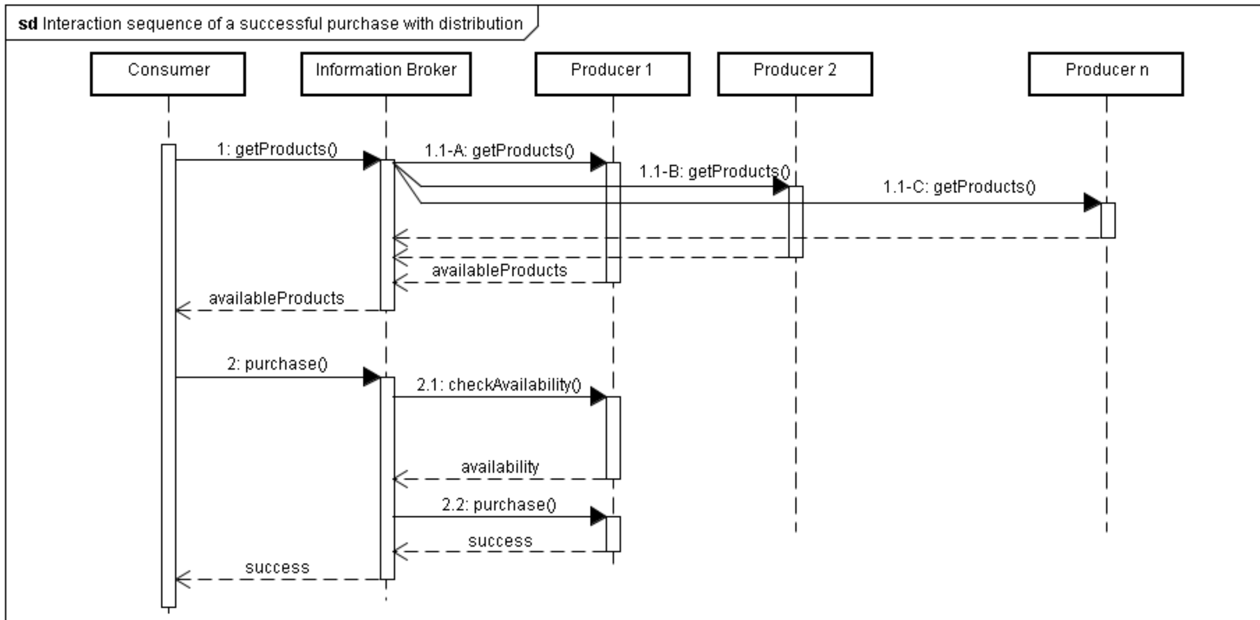


Figure 4: Interaction sequence of a successful purchase with distribution

An interaction sequence of the communication between the information broker and a consumer can be found in Figure 4. For each product information request, connections are made to all the applying producers. For a purchase then, only a connection to the corresponding producer is made.

Since the core business of information brokers is to provide its consumers with information, it is important that this data is accurate. On top of that, the database of an information broker grows continuously with the growth in (detail of the) offers, which increases the risk of a database malfunction. This has motivated information brokers to investigate how information on goods and services offers can be accessed on real-time.

1.2 Objectives

The main objective of this research is to determine the most suitable architecture and technology to realize distributed access on real-time to information on products. There are two sub-objectives to reach this objective:

1. To provide an overview of existing integration technologies;
2. To provide an overview of architectural options to use these integration technologies for an information broker and determine the most suitable one.

There are two more sub-objectives to validate the results of the main objective:

3. To provide an overview of existing implementations of the integration technologies and determine the most suitable one;
4. To create and test a prototype which provides an example implementation for the proposed architecture.

To achieve distributed search and communication with the producers in a uniform way, a certain

degree of cooperation has to be expected from them. In order to make use of external webservices in a distributed way, these webservices need to be available. Some producers may not have such a functionality yet, but provide their information in other ways, for example through an XML file distributed to the information brokers over an FTP connection. How to encourage the producers to provide web services is outside the scope of this research, as we focus on the possible techniques and the feasibility of using these techniques for our purpose.

1.3 Approach

In order to reach the main objective (*to determine the most appropriate architecture and technology to realize distributed access on real-time to information on products*), the following steps have been taken:

1. **Requirements analysis**

   The requirements of an integration solution for an information broker have been identified, and a weight factor has been assigned based on the wishes of the company of our case study. These requirements have been used to compare the possible architectures objectively.

2. **Literature survey of existing technologies**

   Research has been done to find out which technologies are currently available, and the results of this survey have been presented in an overview.

3. **Identification of possible architectures**

   Based on the existing technologies from the previous step, several different possible architectures for an information broker have been identified.

4. **Comparison of possible architectures**

   From the identified candidate architectures, the most suitable one has been chosen.

In order to validate these results, a case study was carried out through at TSi Solutions [45]. TSi is the number one information broker in the travel industry in the Netherlands. For this validation the following steps have been taken:

5. **Literature survey of existing implementations**

   The most popular implementations of the chosen technology have been identified and presented in an overview.

6. **Selection of the best implementation for this purpose**

   Based on our requirements, a Hello World example, and two comparisons from literature, the most suitable one for our case study has been chosen.

7. **Implementation and testing**

   For the previously chosen integration technology implementation, a configuration with supporting software has been created to provide an example of how this architecture can be used in practice. The performance of the configuration and software have been tested under increasing load.

1.4 Report Structure

This report is further structured as follows: Chapter 2 gives the requirements of the architecture and its implementation. Chapter 3 describes and compares currently available integration technologies. Chapter 4 presents several architectures, which make use of the described technologies, compares them and chooses the most suitable one for this purpose. Chapter 5 contains an overview of different implementations of this chosen architecture, and after a comparison, it chooses one or more promising implementations for further study. Chapter 6 covers the implementation of prototypes for the chosen technology or technologies. Finally, Chapter 7 contains our conclusions and discusses some open issues for further work.

2. Requirements

The goal of this chapter is to identify the requirements of a distributed information access system for an information broker. These requirements are used to objectively compare the candidate application integration architectures. This chapter is further structured as follows: Section 2.1 describes the approach taken, Section 2.2 introduces the stakeholders, Section 2.3 describes how the stakeholders will use the integration solution, Section 2.4 contains the functional requirements. Section 2.5 describes the non-functional requirements, and Section 2.6, finally assigns a priority to each of the requirements.

2.1 Approach

In this chapter the requirements for an integration solution for an information broker have been identified. The input for these requirements has been received from three sources:

1. **Case study company analysis**

   The existing architecture and available systems have been analyzed.

2. **Questionnaire**

   A questionnaire has been filled out by the chief technology officer (CTO), and several system architects and developers at the information broker of our case study. The questions from the questionnaire can be found in Appendix A.

3. **Discussions**

   With one of these architects, intensive discussions have been held, based on his experience in the application integration domain.

We have approached the requirements analysis through the following taken steps:

1. **Identification of stakeholders**

   Who may experience benefits or drawbacks of the solution?

2. **Identification of use cases**

   How will the integration solution be used by the stakeholders?

3. **Identification of functional and non-functional requirements**

   Which criteria shall be used to compare the candidate architectures?

4. **Assignment of weight factors**

   Which of the identified criteria have a higher or lower importance?

2.2 Stakeholders

In the results of the questionnaire, stakeholders have been identified at different levels by the different respondents. We have categorized the stakeholders in the following (sub-)categories:

1. **Consumers**

   The consumers are the users of the services offered by an information broker. For an information broker in the travel industry, such as the one from our case study, the consumers could be split up into two sub-categories:

   a) **Travel agencies**

      Some of the consumers own physical travel agencies where their customers can go to get brochures and book a vacation or request advice.

   b) **Website developers**

      Most of the consumers of the information broker's services provide on-line travel agencies which offer a search and book functionality to their customers.

   However, since these sub-categories share the same concerns, namely fast and accurate information access, we decided to regard them as one category.

2. **Information broker**

   The information broker is a high-level category, which can be split up into the following categories:

   a) **System architects**

      System architects are interested in the sustainability of a new architecture.

   b) **Developers**

      In case an information broker is working in a domain without one single open standard on the web service interfaces, transformations between the interfaces will need to be defined. Enrichment of the information will also need to be addressed by developers.

   c) **System administrators**

      A system administrator is interested in the performance under heavy loads and possibilities of extending the computing power, for example through clustering.

   d) **Project leaders**

      Project leaders are interested in the time necessary to add a new producer, or to add a completely new service to the existing ones.

   For the use cases we have considered these sub-categories as one, since there we have addressed how the applications are used rather than how they are developed.

3. **Producers**

   Producers can take different forms for the different services offered by the information broker. For an information broker in the travel industry, one can think of:

   a) Tour operators;

b) Insurance companies;

c) Payment service providers;

d) Car rental providers;

e) Transportation companies (such as airline, railway and bus companies).

However, for the requirements analysis we can generalize these categories though, as they all offer products through web services, and the exact flow or content of purchases is not relevant yet. The services of the producers need to be accessed through the new integration solution, and shall not need to be altered.

In this analysis, we regarded a stakeholder as a role, and not as a specific person. The result of this point of view is that persons or companies may take the role of more than one stakeholder.

2.3 Use cases

Figure 5 shows the use case diagram for the information broker's applications. A consumer specifies which offers shall be available, searches the offers of the producers of his choice (unless the consumer is blocked by the producer), initiates purchases and cancellations, and has the possibility to request information on purchases from the past. A producer shall have the ability to configure which consumers are allowed to use or offer his products, handles the applying purchases and cancellations, and provides information about his products. The information broker's task, finally, is to direct purchases and cancellations to the corresponding producers, distribute requests for product information to the applicable producers and to enrich and transform those responses.

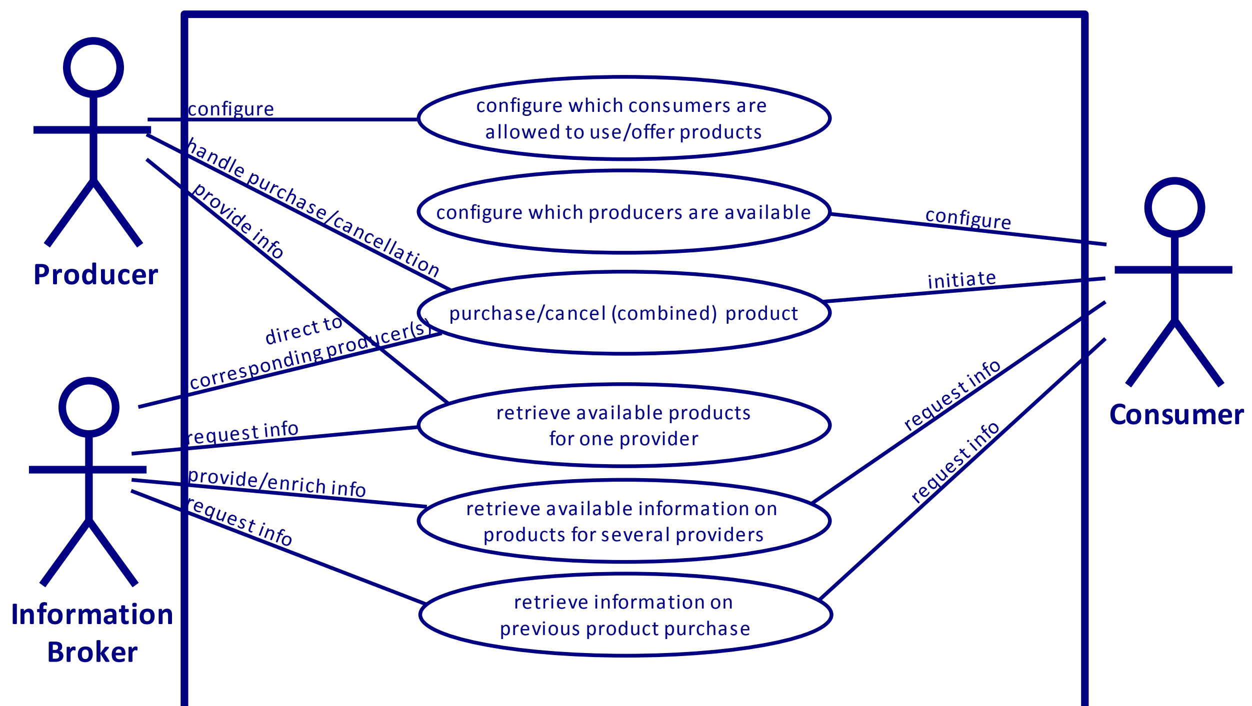


Figure 5: Use case diagram information broker

2.4 Functional requirements

This section lists of the functional requirements and a brief description of their meaning. Functional requirements are those requirements that describe what the solution shall provide, not how or how fast. The questionnaires have provided the input which technologies, such as transports and protocols, shall be supported by the integration solution.

1. **Generic transportation of messages**

   The core business of information brokers is to move information from many sources to many consumers in a few different formats. According to most literature on information brokers, such as [14], this should be one format, but this oversees the possibility of different versions of such a format.

   Due to this relatively small number of formats, it is desirable to keep a strict separation between the message format and protocol used by a specific producer and the other logic for that service. The solution shall therefore provide a generic way to let applications of the information broker communicate with other (possibly external) parties. This leads to the following two sub-requirements:

   a. **Generic support for multiple transports**

      As the number of used applications grows, so does the number of transports used for the communication (HTTP, HTTPS, JMS, etc.). The applications using the integration solution shall be unaware of the used protocol of the applications they are communicating with.

   b. **Generic support for multiple protocols**

      Integration solutions where many different parties are involved, each with their own implementation, shall always be based on standards. The most popular standards for this purpose at the moment are REST, SOAP, XML-RPC, CORBA and DCOM [6].

   c. **Transparent message exchange**

      The applications shall be unaware of the exact structure of the messages (SOAP, XML, CSV, etc.), as they are expected by the other applications. This counts not only the type of message, but also for the naming and presence of elements, attributes, values, order of columns, etc.

      Transformation to the required structure shall be configurable in the integration solution for each communicating party individually. This way new service providers can be added easily, in order to increase the options to choose from for the consumers of the information broker's services.

2. **Add existing applications easily**

   It shall be possible to let existing applications make use of the integration solution without any changes in these applications.

   a. **External applications**

      Applications from third-parties cannot be changed by the information broker, and cannot be expected to be changed by those third-parties either.

   b. **In-house applications**

Applications developed by the information broker can be changed, but it is desirable that they do not have to be.

3. **Routing**

   The solution shall provide a content-based routing mechanism to decide where messages are supposed to be directed to. Some messages will be intended for several producers of a specific type (e.g. tour operators, payment service providers, etc.), and others for a single provider only (e.g. a product purchase). On top of that, this routing functionality shall be able to perform fair load balancing.

4. **Global configuration**

   There shall be a possibility to update all servers of the integration solution configuration with a single application (possibly duplicated in order to prevent a new SPOF). This application shall then distribute the new configuration to all servers in the cluster. This may for example be necessary when a consumer of the information broker's services wants to switch to a newer version of the request and response format.

5. **Security & licenses**

   An information broker generally offers several different services, for which separate licenses are available. Based on the credentials the integration solution is able to find out whether or not this consumer has access to the called service. A consumer can for example have purchased licenses for the search and purchasing services, but not for the payment service, since he created his own implementation for this.

   The integration solution shall therefore provide a solution which does not only check whether or not the request is issued with valid credentials, but also whether or not the sender of the request has a license to that specific web service.

2.5 Non-functional requirements

This section lists non-functional requirements, which are desirable properties of how the integration solution.

1. **Open source**

   Open source products have the benefit of being potentially cheap. Another benefit is the possibility to cooperate with the open source project when a bug is found and needs to be fixed fast from the information broker's point of view, rather than waiting for the supplying company to fix the bug.

   Another important benefit from (successful) open source projects is that they have a strong community supporting them. This allows these projects to mature faster (i.e. contain less bugs) than other products, be supported by many on-line code samples, and a fast on-line support with often more than one response to a problem statement, which creates a 'second opinion' validity of these responses.

2. **Ability to scale out**

   It is necessary to be able to add servers fast in order to cope with growth of the number of consumers or peek loads.

3. **Ability to upgrade the system without a complete shutdown**

   Upgrading to a newer version of the integration solution may discontinue operation for one of the servers in the cluster at a time, but not for all at the same time.

4. **Reliable messaging**

   It shall be possible to use reliable messaging. Ideally this should be configurable per type of message in order to save resources at peek loads.

5. **Independence of implementation language**

   The solution shall be independent of the implementation language used at both communicating parties.

6. **Operation time**

   The minimum operation time of the solution shall be five years.

7. **Message speed**

   As a result of our discussions with the system architect, we have defined the average size of a message for our purpose to be around 5 kilobytes. The company of our case study has required that such a message should be able to do a two-way trip within 50 milliseconds.

8. **Fail-fast adequacy**

   The case study company has also required that when a component fails, it shall take the whole system less than 30 seconds to recover.

2.6 Weighting Factors

Weight values are assigned to all requirements in order to determine their relative importance and priorities, as shown in Table 1. These weighting factors have been provided by a system architect with 14 years of experience in the application integration domain, and are used to compare the candidate architectures objectively.

Requirement	Weight	Justification
Functional requirements		
1. Generic transportation of messages	-	
a. Support to multiple transports	4	It is important that the IB only needs to configure for different transports, rather than writing custom code for this.
b. Support to multiple protocols	4	Just as for transports, it is important that the IB only needs to configure for different protocols, rather than writing custom code for this.
c. Transparent message exchange	4	Transformations from and to the producer's model shall be strictly separated from other code and configurable per producer.
2. Add existing applications easily	-	
a. External applications	5	It is vital that external applications do not need to be changed, as it is impossible to expect this kind of cooperation from the producers.
b. In-house applications	3	It is highly desirable that in-house applications do not need to be changed.
3. Routing	4	It is important that the solution can detect for which producers requests are intended, and sends them there.
4. Global configuration	1	It would be a nice feature if configuration can be done in one existing application.
5. Security	5	It is vital that there is a security mechanism which detects early whether or not a request was sent by a known party, before any transformation is carried out, in order to save resources, against for example DDoS attacks.
Non-functional requirements		
1. Open source	4	It is important that the proposed solution is (at least potentially) cheap, and still provides good support (through a community).
2. Ability to scale out	5	It is vital that clustering is possible, to distribute the load on the integration solution.
3. Ability to upgrade the system without a complete shutdown	4	It is important to have an up-time as high as possible, so system shall not need to be restarted for updates.
4. Reliable messaging	3	It is highly desirable to use reliable messaging for requests such as purchases. This is not

		rated as important, as a workaround could be created for this, by wrapping the called services with a reliable messaging service.
5. Independent of implementation language	5	As the information broker has no influence on which programming language is used by producers or consumers, It is vital that communication with the integration solution is independent of the implementation language.
6. Operation time	4	As an architecture should be sustainable, it is important that this will not need to change again over the next five years.
7. Message speed	4	Since an architecture with distributed information access requires extra communication, it is important that this communication is fast.
8. Fail fast adequacy	3	It is highly desirable that any failing system is dealt with adequately. It is not rated as "important" since a workaround to solve this problem could be created in a relatively small amount of time.

Table 1: Weighting factors for requirements

3. Comparison of Integration Technologies

This chapter presents the currently available application integration technologies. First we discuss which integration technologies to compare, and which approach to take for this comparison. Then we compare the identified technologies. This chapter is further structured as follows: Section 3.1 describes the approach we took to compare the integration technologies, Section 3.2 presents the point-to-point integration, Section 3.3 describes the hub-and-spoke integration, Section 3.4 covers the enterprise message bus integration, Section 3.5 the enterprise service bus integration, and Section 3.6, finally, concludes this chapter.

3.1 Approach

In our problem situation, the external web services of the producers need to be integrated in the information broker's architecture. Several integration solutions have been discussed in existing literature, such as [2], [9], [10]. The latter has listed them as following:

1. Point-to-Point Integration
2. Hub-and-Spoke Integration
3. Enterprise Message Bus Integration
4. Enterprise Service Bus Integration

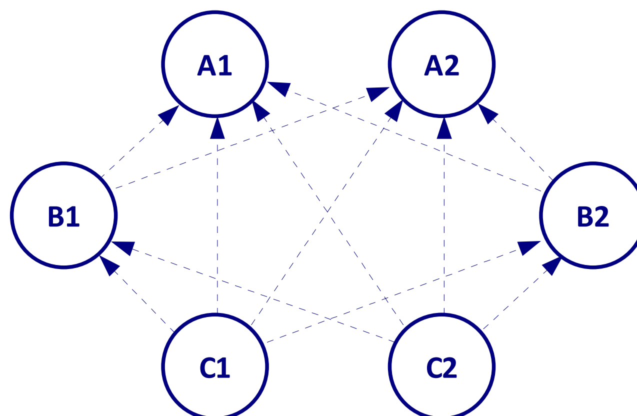


Figure 6: Situation before integration

We compared these technologies, by assuming three applications. Each of these applications have been duplicated on two servers to cope with heavy loads. The schematic overview of the system before choosing an integration solution can be found in Figure 6. In this figure nodes with the same prefix imply servers with the same application. The dashed lines in Figure 6 indicate information needs. In this example, application C needs information from both A and B, and application B needs information from application A. This diagram is elaborated on throughout this chapter, by adding continuous-lined arrows that indicate requests over network connections (where the point of the arrow indicates the direction of each request).

3.2 Point-to-Point Integration

Software projects normally start with point-to-point integration, as it is the most intuitive and fastest way of connecting two communicating parties. Point-to-point integration means that a connection is created for every pair of parties who are interested in each other's information, as depicted in Figure 7.

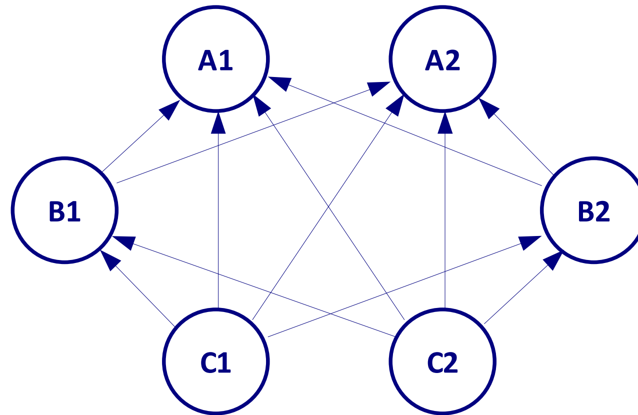


Figure 7: Point-to-point integration

The main drawback of point-to-point integration is the poor maintainability. As the number of communicating parties increases, the number of connections increases exponentially. An interface change at one of the parties forces all other parties which communicate with it to change their implementations, making it too difficult to maintain.

3.3 Hub-and-Spoke Integration

The hub-and-spoke model was intended as an improvement upon the point-to-point model and finds its origin in the airline industry. Delta Airlines claims it was the pioneer of this model in 1955 [11]. Since then, the hub-and-spoke model has been applied to shipping, overnight express delivery, and many other activities of transportation [44], for example by FedEx [41].

The main principle of the hub-and-spoke model is to have a relatively small amount of central points (hubs). These central points are connected to many, if not all, of the other servers. In our example we can achieve this in two ways:

1. **Single hub Configuration**

   A new server is introduced which is responsible for all the communication. This new server (N in Figure 8) is called the hub, and the connections to the other servers are called the spokes. In our example, the connections between N and the B-servers are bidirectional, since N can issue a request to B in the name of application C, but N can also be used to issue a request to A in the name of B. B can therefore be the caller and the callee in the communication with N.

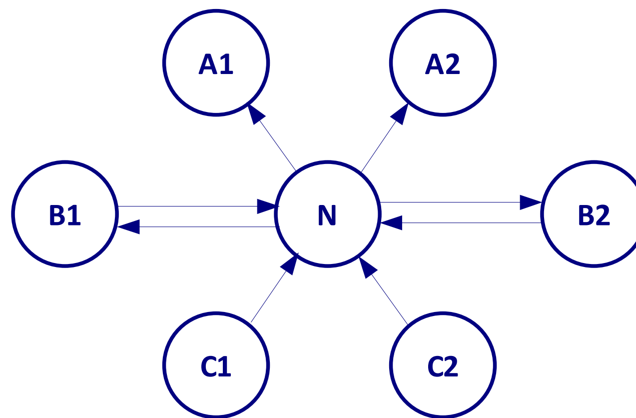


Figure 8: Hub-and-spoke with new central server

2. **Multi hub Configuration**

   The existing servers with the prefix B can also be used as hubs. In the case illustrated in Figure 9, server B1 is responsible for the communication with the A-servers, while server B2 handles the requests by the C-servers, possibly by simply forwarding them to B1.

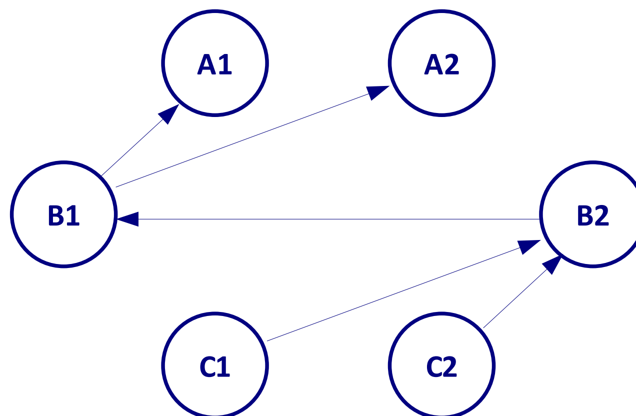


Figure 9: Hub-and-spoke with two central servers

The hub-and-spoke model has been adopted in the software engineering domain by traditional Enterprise Application Integration, which has a strong tendency towards the single hub configuration. The advantages of the hub-and-spoke model are the following:

- **Small amount of connections**

  In the strict application of the hub-and-spoke model, the number of connections can be reduced to n-1 (where n is the number of servers), compared to the worst-case n*(n-1) for point-to-point integration.

- **Single communication protocol**

  With a hub-and-spoke technology, developers need to consider one communication protocol only, rather than one for each related application, as for the point-to-point solution [13].

- **Easier to switch applications**

  Applications are unaware of which applications are called upon by the hub.

The drawbacks of this model are for both the single hub and the multi hub configuration:

- **Poor scalability**

  In [13] the lack of scalability is addressed as the major problem with the hub-and-spoke model. The reason for this is that all information from applications has to be processed or passed on by a single hub server, for the single-hub and multi hub configuration respectively. This causes the hub to become a bottleneck for the system.

- **Single point of failure**

  One of the motivations for this research was to eliminate the SPOF introduced by the searches on the database, but having a single hub would reintroduce this problem in a different place. [13] This applies even more to the multi-hub configuration, where multiple SPOFs are introduced.

3.4 Enterprise Message Bus Integration

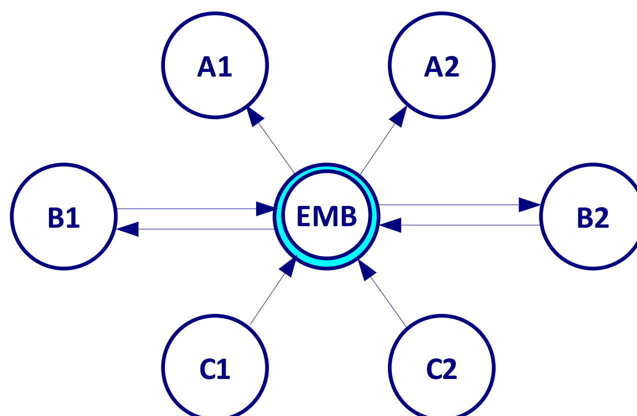


Figure 10: Enterprise Message Bus

An Enterprise Message Bus (also known as a *Message Queue*) is a message channel consisting of several message servers. Figure 10 shows that this solution is somewhat related to the hub-and-spoke model. However, there is a significant difference, as the central 'point' (the *bus*) consists of multiple servers, whose sole purpose is to pass on the messages to the applicable recipient. These servers are completely unaware of the content of the messages passing through them.

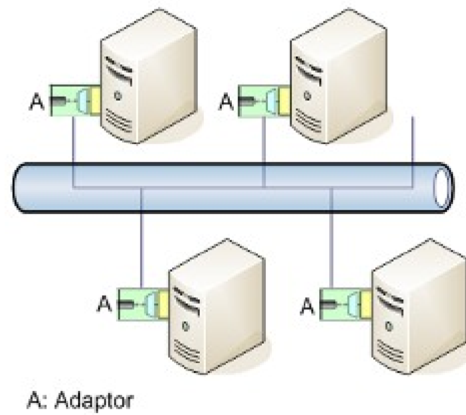


Figure 11: Internal structure of an EMB [7]

Figure 11 shows how ESBs are internally organized. For an Enterprise Message Bus, adapters are located at the applications that use the bus. These adapters translate messages from the *canonical model* (the model used for communication on the bus) to the *specific model* for that application, and vice-versa.

Message-Oriented Middleware
Message-Oriented Middleware (MOM) is software which provides a solution for messaging. Every server that runs this software can guarantee that messages are delivered once and only once, by persisting the message at least until its arrival has been confirmed. On top of that, it can enqueue messages, so that the server that accesses them (which may very well be another message server), can decide when to request the next pending message. In general, there are two types of messages:

- **Publish-and-subscribe**
  Publish-and-subscribe messaging implies that a message is directed to all those who have subscribed to the topic. There may be zero or more subscribers, and the message server publishes the message to all of these. An example with three subscribers is given in Figure 12, where a solid line means a delivered message. The sender sends a message to the middleware, and the middleware directs this to the three receivers.

  For this type of message, generally the receivers perform either different tasks (e.g. book an airplane ticket, book a hotel, and book a rental car for the same complete booking), or the same tasks in a different environment (e.g. call three different external web services to find product information).

Figure 12: Publish-and-subscribe mechanism

- **Single recipient**

Single recipient messaging implies that a message is directed to exactly one subscriber. An example for this is given in Figure 13, where a solid line indicates a delivered message again, and a dashed line a message which is never sent. The sender sends a message to the middleware again, but with this type of message, the middleware choses one of the three candidate receivers, while ignoring the other two for the moment. How this choice is made depends on the implementation. For this type of message, generally the receivers perform the same task in the same way, but run next to each other to cope with heavy loads.

Figure 13: Single recipient

The advantages of the Enterprise Message Bus are the following:

- **Scalability**

  The bus can be extended as the network extends, by adding extra message servers. Next to that the applications which are accessed the most can be duplicated on multiple servers, and approached through the single recipient message type.

- **Single Point of Failure can be avoided**

  By running duplicates of all application servers, and at least two message servers, any application can fail, without the entire system failing. Although performance may suffer from a missing server, the system can still keep running.

- **Easy to add servers**

  Contrary to the hub-and-spoke model, it is even possible to easily add servers which represent the 'hub', since the 'hub' is now connected to the bus in the same way as the producers and consumers. The consumers call a service of the information broker over the bus.

- **Security layer**

  Many EMBs such as Apache ActiveMQ [1] and IBM WebSphere MQ (WebSphere, 2009) provide a security layer.

The drawback of this technology is:

- **No message transformation**

  EMBs do not provide message transformation mechanisms, although this is one of the requirements for the integration solution, due to the mismatches between the interfaces of the different producers.

3.5 Enterprise Service Bus Integration

In the ESB architecture there is a backbone, consisting of one or more message servers. This backbone functions as a message channel, just like with the Enterprise Message Bus, as can be seen in Figure 14. However, an Enterprise Service Bus does more. Four key components of an ESB

have been identified in (Schulte, 2003): MOM (as discussed previously), web services, XML transformation, and intelligent routing. XML transformation can be done through XSLT or other languages with transformation capabilities. The routing is *'intelligent'* in the sense that the *routers* or *connectors* contain logic to bind to services at run-time, rather than statically to a specified address [10].

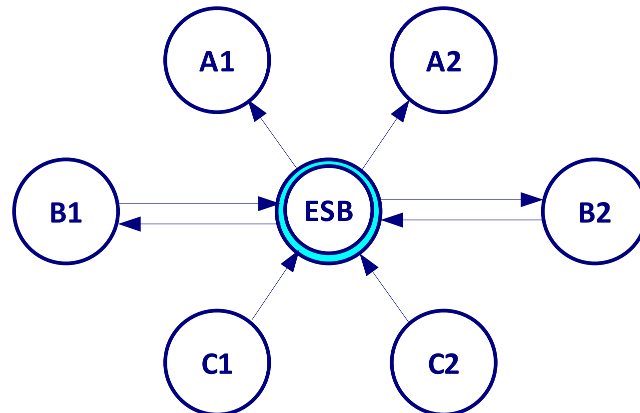


Figure 14: Enterprise Service Bus

The place of this *intelligent connector* is not eminently clear from Figure 14 yet, but can be seen by considering the internal structure of ESBs in Figure 15. Rather than a local adapter, as is used for the EMB, an ESB provides applications with a connector on the bus. ESB is in fact an extension of EMB; while an EMB solely routes messages to their destination, an ESB also transforms messages (between the canonical model and the producer-specific model), and also detects where the destined service of the messages is currently available.

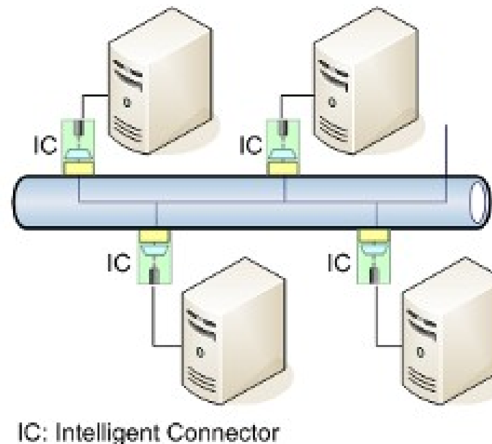


Figure 15: Closer look at ESB [7]

The service container of an ESB is the point where applications, files, databases and other information sources are turned into providers or consumers of services [10]. A specification of these service containers has been developed under the Java Community Process: the Java Business Integration specification JSR 208 (JCP, 2005).

The advantages are similar to the ones for the EMB. However, an ESB provides some more functionality, such as service discovery, message transformation and a security layer is specified in the JBI specification [36].

3.6 Conclusion

Point-to-point integration has been replaced by the hub-and-spoke model a long time ago, and has proven to be hard to maintain as the number of communicating applications grows. Therefore, we will no longer consider the point-to-point integration architecture as a plausible solution.

EMB and ESB are closely related, but ESBs provide a transformation functionality on the bus, which in our problem scenario, especially with the requirements of message transformation and not changing the external parties' applications, is an important difference.

4. Application Integration Architectures

This chapter discusses how the technologies described in the previous chapter can be used in architectures for application integration for an information broker. At the end of the chapter a choice is made which architecture is most suitable to solve our problem, based on the previously defined criteria. Section 4.1 presents a hub-and-spoke architecture, Section 4.2 describes three options to use an Enterprise Message Bus, Section 4.3 discusses the same options, but making use of a Enterprise Service Bus instead, and Section 4.4, finally compares all the alternatives and makes a decision based on the requirements of Chapter 2.

4.1 Hub-and-Spoke Architecture

For the hub-and-spoke model, the single hub configuration suits our problem scenario best, in order to save resources and not slow down communication by adding more intermediate steps than necessary. Due to the extendability requirement for the integration solution, multiple servers need to be running next to each other, all functioning as a single hub. Therefore, a fair load balancer needs to be used as the contact point for the consumers, as depicted in Figure 16. This load balancer then directs the requests from the consumers to the hubs (depicted as the large servers). The hubs are responsible for calling the applicable information broker's services, which are depicted as the small servers. Communication between these services and the external services of the producers takes place through the hub again, as specified by the hub-and-spoke model. This two-way communication is illustrated by the double arrow between the services and the hubs.

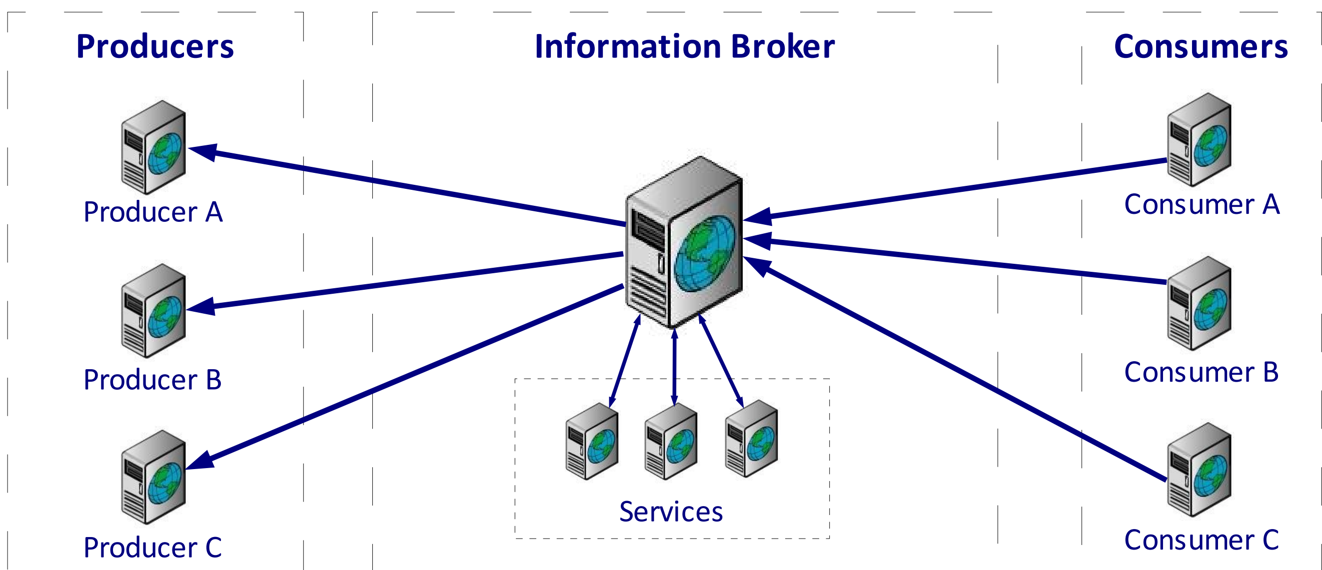


Figure 16: Hub-and-spoke model for the integration broker

4.2 Enterprise Message Bus Architectures

An Enterprise Message Bus can be used to communicate with:

- producers only;
- consumers only;
- both producers and consumers.

As the new integration solution shall be used for the communication between the information broker and the producers, all the three candidate architectures in this section will use an EMB for this communication. For the communication between the consumers and the information broker we then have the following three options:

1. Communication through a load balancer;
2. Communication over a separate EMB;
3. Communication over the same EMB.

These three options will be treated in the respective subsections.

4.2.1 Bus combined with a load balancer

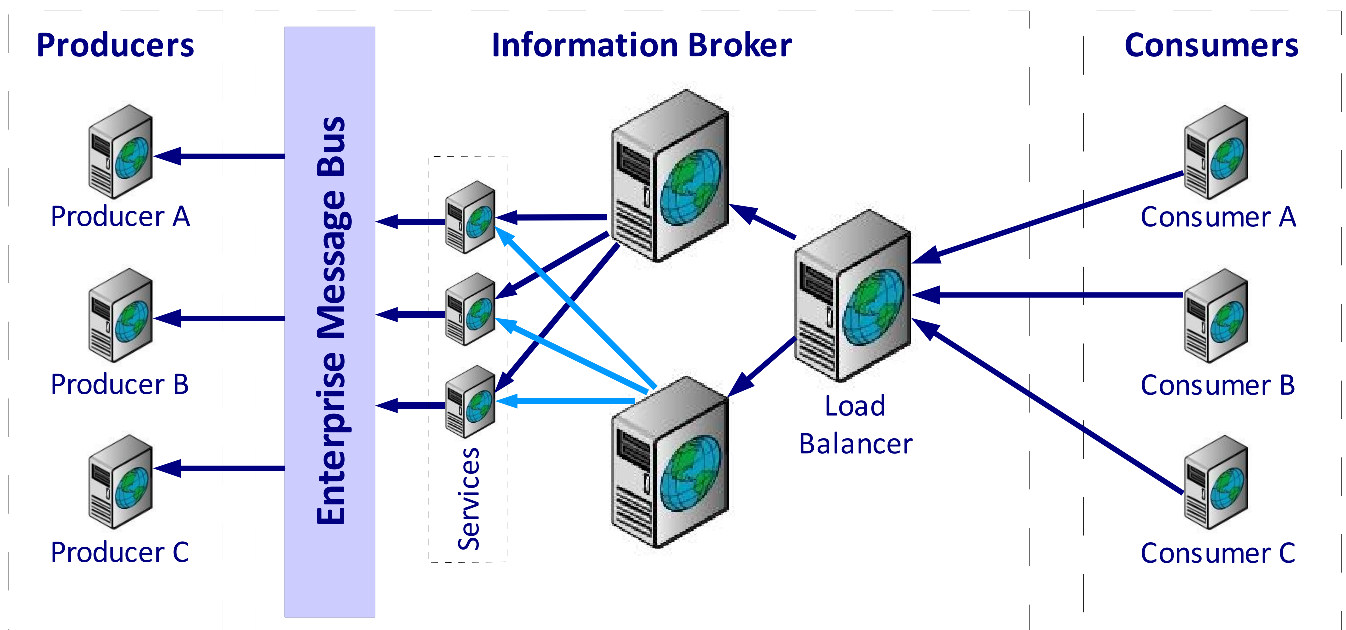


Figure 17: Enterprise Message Bus for communication with producers only

Using an Enterprise Message Bus for the communication between the information broker and the producers enables the services to contact the producers in a distributed way. Consumers, however, still reach the services of the information broker in a point-to-point fashion, as is the current situation in our case study. These consumers direct their requests to a load balancer, as depicted in Figure 17. The load balancer redirects the requests to the main application, which is responsible for amongst others security and calling the applicable services of the information broker. These services contact the producers through the EMB.

4.2.2 Two buses

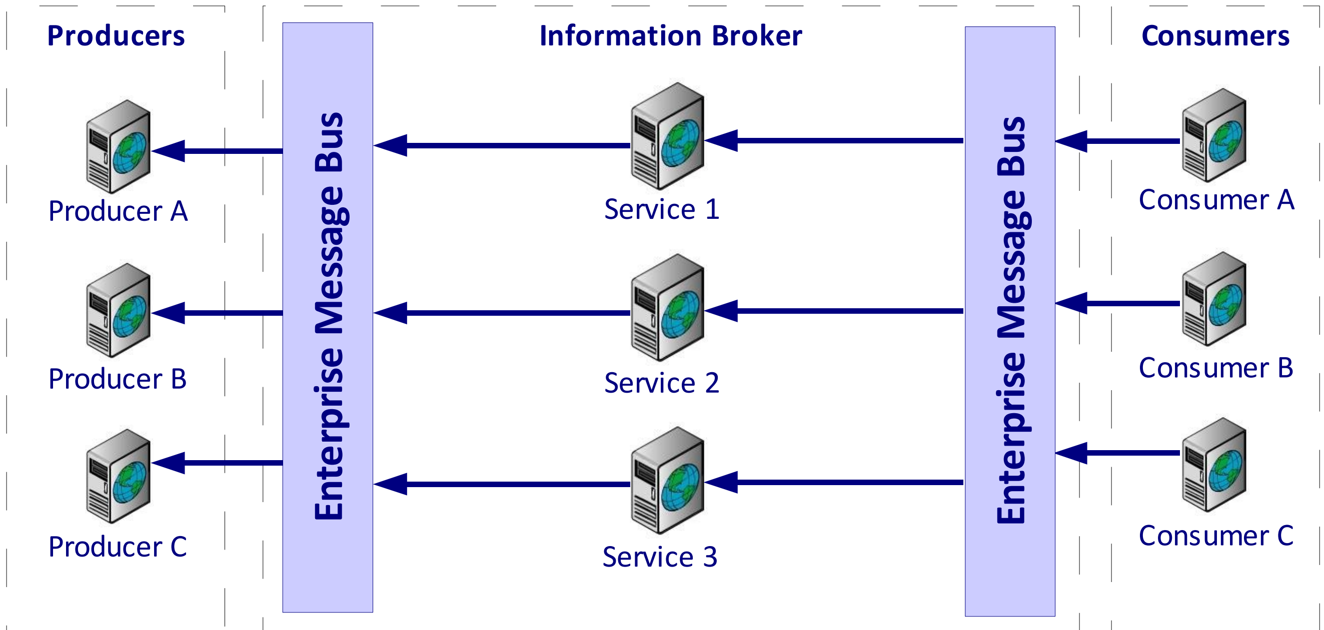


Figure 18: Two EMBs for separate communication with producers and consumers

Another option is to use one EMB for the communication between the information broker and the producers, and another one for the communication between the information broker and the consumers. In this architecture, the tasks of both the load balancer and the main application have been taken over by the message bus, as is illustrated in Figure 18. The advantage of such a structure, rather than using a single bus, is that it prevents consumers from communicating with the producers directly, by physically separating the communication channels.

4.2.3 One bus

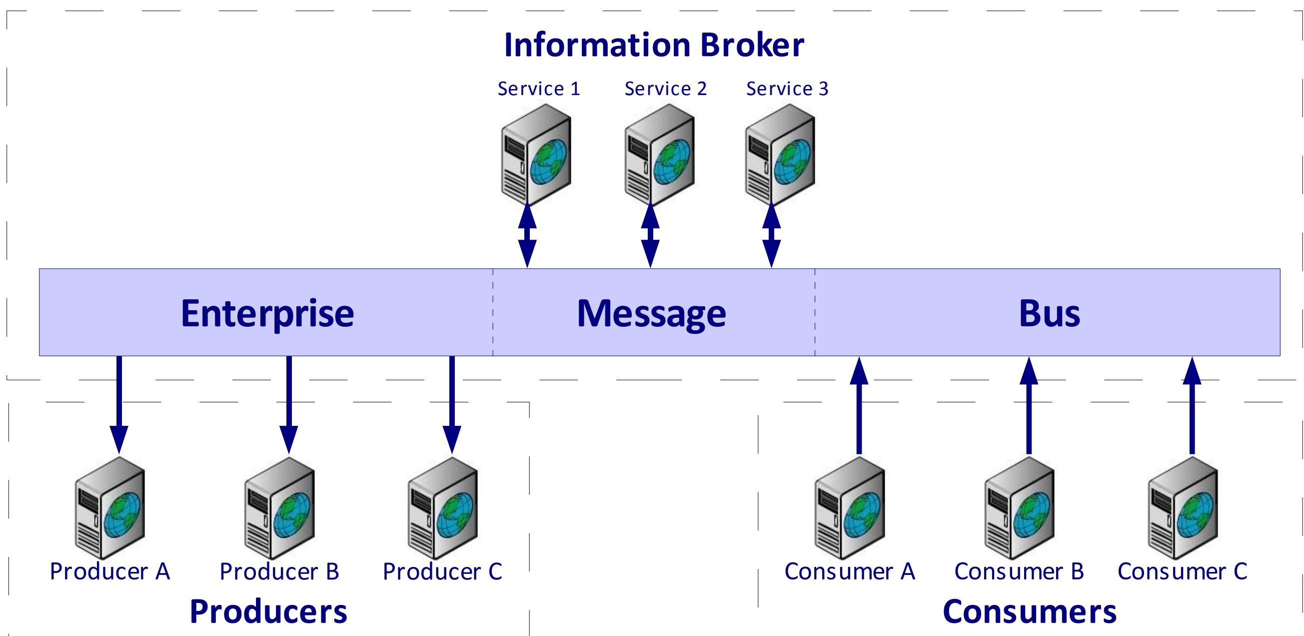


Figure 19: One EMB for all communication

A third option to use an EMB is to use a single EMB for all communication, as illustrated in Figure 19. In this case the bus is extended to include consumers, increasing the maintainability of the entire integration. As depicted by the dashed lines, logical partitioning is necessary to avoid that consumers and producers communicate directly with each other. As seen in the figure, the consumers access the services of the information broker over the bus. The services then contact the applicable producers over the same bus, illustrated by the double arrow between the services and the bus.

4.3 Enterprise Service Bus Architectures

Just as was the case for the EMB, three candidate architectures are presented which all use an Enterprise Service Bus for the communication between the information broker and the producers. The options for communication between the consumers and information broker are discussed in the respective subsections again:

1. Communication through a load balancer;
2. Communication over a separate ESB;
3. Communication over the same ESB.

Although the architectures from this section look the same as their EMB equivalents, the connections are different, due to the mentioned differences in internal structure between these two integration solutions.

4.3.1 Bus combined with a load balancer

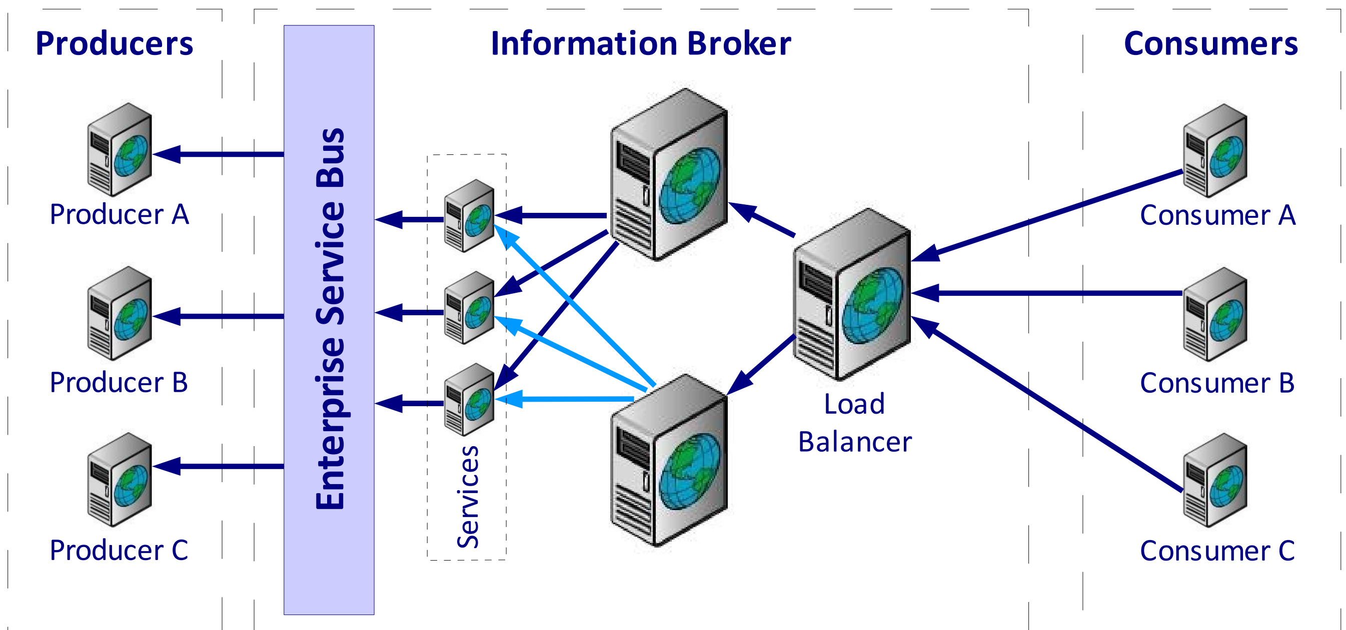


Figure 20: Enterprise Service Bus for communication with producers only

Figure 20 shows the architecture which uses an ESB for the communication between the information broker and the producers. The information broker's services are capable of connecting to the producers in a distributed way. The consumers still connect to the composite services

through the load balancer, and the applicable composite service connects to the respective sub-services. Just as for the EMB equivalent, these composite services are also responsible for the security. The sub-services can contact the producers through the ESB.

4.3.2 Two buses

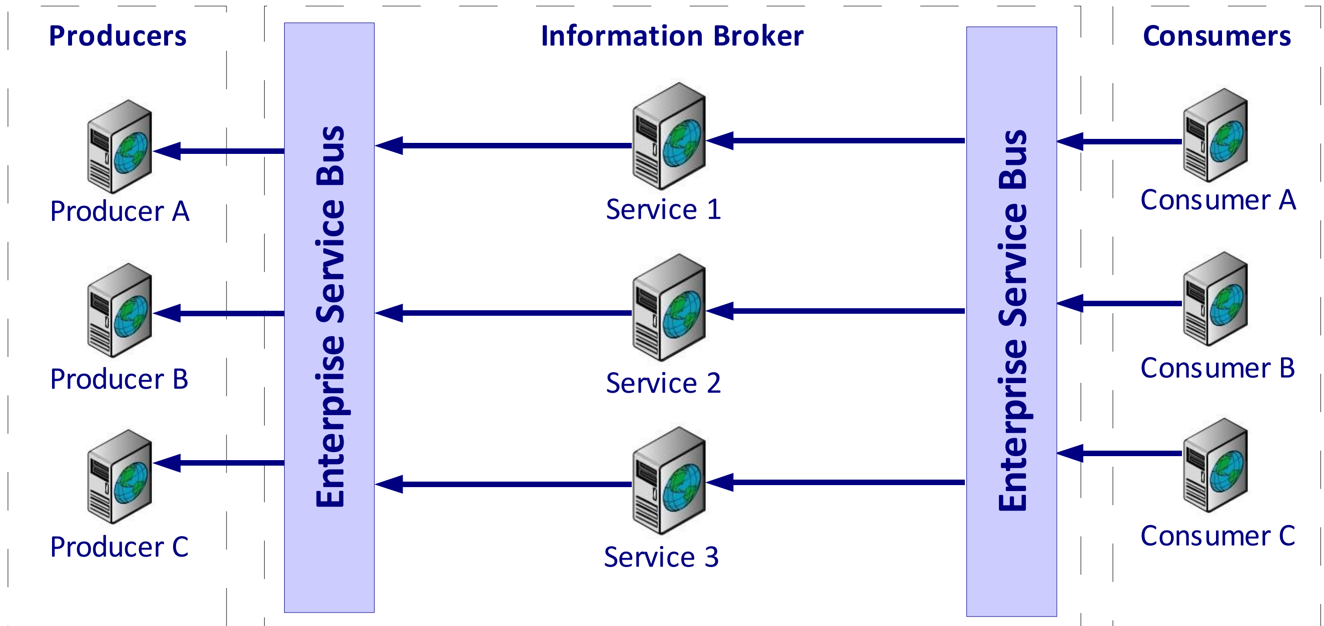


Figure 21: Two ESBs for separate communication with producers and consumers

Just as is possible with the EMB, the ESB can be used on both sides of the information broker as well. One ESB is used for the communication between the information broker and the producers, and another one for the communication between the information broker and the consumers. The tasks of the load balancer are taken over by the service bus, as is illustrated in Figure 21. The advantage of such a construction over one single bus is that consumers can be prevented from communicating with the producers directly by physically separating the communication channels.

4.3.3 One bus

Using a single Enterprise Service Bus for both communication with the consumers and with the producers, as depicted in Figure 22, is actually the way these systems were originally intended to be used. With this architecture, not only consumers and producers can be added easily, but also extra servers for the information broker's services. As depicted by the dashed lines, logical partitioning is necessary to avoid that consumers and producers communicate directly with each other, just as for the EMB. As seen in the figure, the consumers access the services of the information broker over the bus. The services then contact the applicable producers over the same bus, illustrated by the double arrow between the services and the bus.

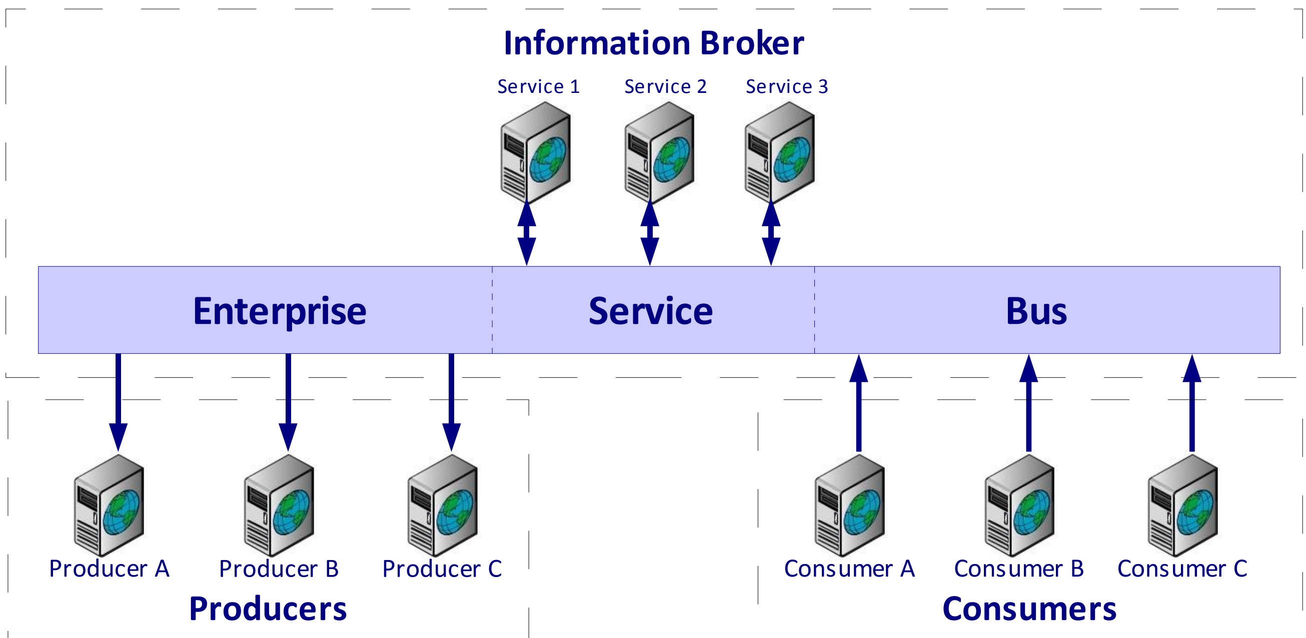


Figure 22: One ESB for all communication

4.4 Comparison

At this stage, we are not yet able to judge whether these architectures meet all our requirements, as compliance to some of these requirements depends on the implementation. Therefore, we had to take a subset of the requirements, and for each of these requirements assign a value to the seven previously discussed architectures. Since it is nearly impossible to use a nominal scale to grade these requirements, we have graded each architecture for each requirement according to the following ordinal scale:

2 points - This architecture meets this requirement.

1 points - This architecture supports this requirement with a small workaround.

0 points - This requirement cannot be achieved with this architecture.

The score of an architecture finally, is then computed through the following formula:

$$score = \sum_{req \in Reqs} (WF_{req} * P_{req})$$

where *Reqs* is the set of relevant requirements, WF_{req} is the weight factor of a requirement from that set, and P_{req} is the number of points obtained for that requirement by that architecture.

The scores for the individual categories can be found in Table 2, where the numbers of the architectures in the header of the table correspond with the numbers of the subsections for the EMB and ESB technologies.

The scoring of the candidate architectures in Table 2 have been based on the following arguments:

Functional requirements

1. **Generic transportation of messages**

   a. **Generic support for multiple transports**

      All the architectures are capable of providing generic support for different transports, since the applications do not access external parties directly. Therefore all architectures have been awarded the maximum amount of points.

   b. **Generic support for multiple protocols**

      Only the core of the architectures (the hub or the bus) is responsible for dealing with the different protocols. Therefore all architectures have been awarded the maximum amount of points.

   c. **Transparent message exchange**

      According to our definitions of hub-and-spoke and EMB, no transformation takes place at the core of the architecture. Transformation takes place at the server the application is running on, implying that messages are not exchanged transparently. Therefore the hub-and-spoke and EMB architectures have not been awarded any points. For ESBs this transformation functionality exists on the bus and can be defined per provider.

2. **Add existing applications easily**

   a. **External applications**

      For the hub-and-spoke model and EMB we have defined that a lightweight connector needs to be present on the server the application is running on, to connect to the bus. This integration difficulty implies that no points have been awarded to these four architectures for this requirement. For ESBs the intelligent connector is located on the bus, allowing applications to be added easily.

      The first ESB architecture still uses a load balancer as a workaround for the communication between the information broker and the consumers, and is therefore only awarded with one point.

   b. **In-house applications**

      For in-house applications adding such connectors on the servers the applications are running on is possible. For the architectures with two buses, consensus needs to be reached which bus is to be used for internal communication. These architectures therefore have only been awarded one point.

3. **Routing**

   All the used technologies provide routing mechanisms, and therefore all architectures have been awarded the maximum amount of points.

4. **Global configuration**

   All of the architectures could be configured through an application as described in the requirement. Whether or not this is present in the solution depends on the implementation.

5. **Security & licenses**

   All the proposed architectures provide a central place where security and licensing issues can be dealt with, and have thus been awarded with the maximum amount of points.

Non-functional requirements

1. **Open source**

   For the hub-and-spoke model, no off-the-shelve open source integration products exist. Open source projects for this purpose have either have been discontinued (such as Business Integration Engine (BIE, 2009)) or evolved into ESB projects (such as Jitterbit [20] and Openadaptor [30]). For the other two technologies, plenty of open source alternatives are available, such as ActiveMQ [1] for EMBs and ServiceMix [39] and Mule [28] for ESBs.

2. **Ability to scale out**

   The hub and spoke model is unsuitable for scaling out, as even in the multi-hub configuration, the throughput is limited by the hub where the original request was intended. The bus architectures are well-known for their scaling capabilities, as for example discussed in [35].

3. **Ability to upgrade the system without a complete shutdown**

   Since EMB and ESB are able to scale out, this can be used temporarily to update the servers in the cluster one by one. For the hub-and-spoke model this is not possible.

4. **Reliable messaging**

   Reliable messaging is supported by many EMBs and ESBs, such as the previously mentioned ActiveMQ, ServiceMix and Mule. For the hub-and-spoke model a WS-ReliableMessaging implementation should be found, such as the Apache project Sandesha [37].

5. **Independence of implementation language**

   All three architectural patterns support independence of implementation language, for example by sending XML or SOAP messages over an HTTP network connection.

6. **Operation time**

   The tendency in the current market is to move away from the hub-and-spoke model, towards ESBs, as discussed in [9], [16], and [25]. It is therefore more likely that ESB architectures will have a longer operation time. As ESBs are in fact EMBs with an extra layer on top, as discussed in [50], ESBs are the most likely technology to have a long operation time.

7. **Message speed**

   The speed of the messages through the integration solution depends on the used implementation.

8. **Fail-fast adequacy**

   The adequacy of the fail-fast functionality depends on the implementation.

As can be seen in the last row of Table 2, the single ESB for communication with both producers and consumers is the most suitable integration architecture for our case.

		H&S	EMB			ESB		
Requirement	**Wt.**	**1**	**1**	**2**	**3**	**1**	**2**	**3**
Functional requirements								
Generic transportation messages	-							
Support to multiple transports	4	2	2	2	2	2	2	2
Support to multiple protocols	4	2	2	2	2	2	2	2
Transparent message exchange	4	0	0	0	0	2	2	2
Add applications easily	-							
External	5	0	0	0	0	1	2	2
In-house	3	2	2	1	2	2	1	2
Routing	4	2	2	2	2	2	2	2
Global configuration	1	Depending on implementation						
Security	5	2	2	2	2	2	2	2
Non-functional requirements								
Open source	4	0	2	2	2	2	2	2
Scale out	5	0	2	2	2	2	2	2
Upgrade system without shutdown	4	0	2	2	2	2	2	2
Reliable messaging	3	2	2	2	2	2	2	2
Independent of impl. lang.	5	2	2	2	2	2	2	2
Resilient to changes	4	0	0	0	0	2	2	2
Message speed	4	Depending on implementation and hardware						
Fail fast adequacy	3	Depending on implementation and hardware						
Total								
Final score		**56**	**82**	**79**	**82**	**103**	**105**	**108**

Table 2: Requirements compliance of the architectures

5. Choosing an Implementation

In this chapter an implementation for the ESB is chosen for our case study. First three open source ESBs with a high market visibility are introduced. In [36] Apache ServiceMix and Mule are rated as the top two ESBs. We have decided to add OpenESB to our comparison, as this implementation is an initiative of Sun Microsystems. The rest of this chapter is structured as follows: Section 5.1 introduces the comparison approach, Section 5.2 describes Mule, Section 5.3 discusses Apache ServiceMix, Section 5.4 introduces OpenESB and Section 5.5, finally, compares these three implementations, based on the still open requirements from the previous chapter.

5.1 Approach

In this chapter three ESB implementations are introduced with the focus on the following points:

1. **Tool Support**

   Which tools are available to support the developers?

2. **Components**

   Which components are available for amongst others transfer protocols, routing, transformation and security?

3. **Hello World example**

   What needs to be done to set up a first, simple project?

In the previous chapter we have left three requirements open for discussion at the implementation level:

1. Global configuration
2. Message speed
3. Fail fast adequacy

At the end of each section, each implementation will be assessed on these three points. These assessments are combined in one proposed implementation at the end of this chapter.

5.2 Mule

Mule is a lightweight Java-based messaging framework [28]. Since it is Java-based, Mule supports Java classes to be used to extend the functionality of the bus. Mule is available in a few different versions, Mule JBI (the JBI compliant version), Mule Community Edition (free), and Mule Enterprise Edition (including support and higher availability).

5.2.1 Tool Support

Mule IDE 2.0 is a development and testing environment based on Eclipse [28]. It supports the following features:

- Create a new Mule project in Eclipse
- Create a working Mule project in Eclipse based on Mule examples

- Create a new Mule configuration file with selected namespace declarations
- Edit Mule configuration files
- Use of Mule Schema auto-completion and documentation (using Eclipse-compatible XML editor with context sensitive content assistant)
- Switch between Mule versions in a project
- Run the Mule Server in Eclipse to test a Mule project
- Debug a Mule project in Eclipse

However, during the creation of the Hello World example, we have experienced this tool to not be entirely free of bugs. The most obvious bug was the "Run as..." menu for the configuration files, which sometimes refused to open until Eclipse (Ganymede) was restarted. Furthermore, no visual tools or wizards are available for the creation of services.

5.2.2 Components

Mule has binding components for around 50 protocols [49] including JMS, JDBC, TCP, UDP, multicast, HTTP, servlet, SMTP, POP3, file, XMPP, SOAP [28]. Furthermore it has (amongst others) message routing capabilities [12], a transformation layer [36], Integrated Security Management and Spring integration.

5.2.3 Hello World example

To compare the structure of the configuration files for the different ESBs, we configure every ESB to receive SOAP requests, and then echo its content as a response. The request and response messages which we have sent or received respectively, can be found in Appendix B. In these Hello World examples, the service is actually *on* the bus. Another option would be to define an outbound endpoint to connect to an external service.

Based on the simple Java class from Listing 1, Mule uses JAX-WS to generate a WSDL file. As it can be seen in the code, the method does nothing else than returning the input. For more information on the JAX-WS annotations, please refer to [18].

```
package com.traserv.research.esb.helloWorld;

import javax.jws.*;

@WebService
public class HelloWorld {

	@WebMethod(operationName="helloWorldOperationRequest")
	public @WebResult(name="output")String
		helloWorldOperationRequest(@WebParam(name="input")String input) {
		return input;
	}
}
```

Listing 1: Java Class for Hello World examples

Using JAX-WS is not mandatory in Mule; other options are Axis, or even a simple Java class to generate the WSDL from, but we have chosen for this implementation, as it gives us some freedom to manipulate the WSDL generation, for example to give the return element of an operation a specific name through the WebResult annotation.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://www.mulesource.org/schema/mule/cxf/2.2"
      xsi:schemaLocation="
            http://www.mulesource.org/schema/mule/core/2.2
            http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
            http://www.mulesource.org/schema/mule/cxf/2.2
            http://www.mulesource.org/schema/mule/cxf/2.2/mule-cxf.xsd">

  <model name="helloWorld">
    <service name="HelloWorld">
      <inbound>
        <cxf:inbound-endpoint
            address="http://localhost:8080/helloWorldService"
            synchronous="true"
            frontend="jaxws"
            namespace="http://esb.research.traserv.com/helloWorld"
            serviceClass="com.traserv.research.esb.helloWorld.HelloWorld"/>
      </inbound>
    </service>
  </model>
</mule>
```

Listing 2: Mule Configuration for Hello World example

The Mule configuration to publish this service can be found in Listing 2. The `inbound` and `outbound` (not used in this example) element are used by Mule to separate the inbound from the outbound endpoints. Endpoints are those points where applications connect to the bus, such as a directory, an email server, or, as in this case, a web service. CXF mules plug-in that supports SOAP over HTTP.

5.2.4 Requirements compliance

Mule complies to the still open requirements as following:

1. **Global configuration**

   Mule Galaxy, a governance solution for Mule, provides a global configuration application [4].

2. **Message speed**

   In [24] it is claimed that Mule has a higher performance than any other implementation.

3. **Fail fast adequacy**

   Failure behavior can be configured to a certain extent in Mule. An exception based router can be used to specify Mule's behavior in case of an exception. For example Mule can be configured to access several endpoints until the message has successfully been delivered [4].

5.3 Apache ServiceMix

Apache ServiceMix is based on the Java Business Integration (JBI) specification JSR 208 [39], and has been released under the Apache license. Because Apache ServiceMix is based on the JBI, it

conforms to the JBI configuration files. According to [4] this causes the initial learning curve to be steeper.

5.3.1 Tool Support

ServiceMix is a typical Apache project, which can be managed via command line in combination with Maven2 configuration files [3]. Plug-ins for Maven2 are also the only ones bundled with ServiceMix. The service assemblies and their dependencies to service units can be managed with Maven2. We highly recommend some in-depth knowledge of Maven before starting to create services for ServiceMix.

5.3.2 Components

ServiceMix provides binding components for a file system, an FTP server, HTTP messages, JMS, email and SOAP messages. Besides that there are service engines available (amongst others) for: Plain Old Java Objects (POJOs), (content-based) router or services, filter, splitter, content enricher, scheduler, message transformations and scripting (Ruby, Perl or Groovy).

5.3.3 Hello World example

Configuration of ServiceMix is possible in two ways: a static configuration, or by deploying an archive to the `hotdeploy` directory of the server installation. The latter is the one which is recommended by the developers, and conforms to the JBI specification. We have found it quite complicated to set up a project for the first time, especially because the provided examples do not work in the binary distribution.

The configuration is spread out over several files. The WSDL file can be found in Appendix C. A resource on the classpath is also needed with the name `xbean.xml`, which can be found in Listing 3. The first bean refers to the same POJO web service implementation from Listing 1. The second bean defines the endpoint to the outside world.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:hw="http://esb.research.traserv.com/wsdl/HelloWorld/helloWorld"
       xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0">

  <jsr181:endpoint pojoClass="com.traserv.research.esb.helloWorld.HelloWorld" />

  <http:endpoint service="hw:helloWorldService"
                 endpoint="soap"
                 targetService="hw:helloWorldService"
                 role="consumer"
                 locationURI="http://localhost:8080/HelloWorld/"
                 defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
                 soap="true" />

</beans>
```

Listing 3: Bean definition for Apache ServiceMix configuration

With Maven a file called `jbi.xml` can be generated through the jbi-service-assembly plug-in. This file is packed together with the .jar archives containing the (generated) Java classes and bean

definitions. However, setting up this structure properly requires a substantial amount of experience with Maven. Copying the archive generated by Maven to the `hotdeploy` directory of the running ServiceMix distribution starts the service right away.

5.3.4 Requirements compliance

ServiceMix complies to the still open requirements as following:

1. **Global configuration**

   As ServiceMix is administered from the command line, no global configuration application is available. In the test scenario from [4], some parts of configuration updates were not recognized and needed a restart of the server.

2. **Message speed**

   According to [3], ServiceMix' performance is inferior to OpenESB's.

3. **Fail fast adequacy**

   In ServiceMix, BeanFlow can be used as an alternative to BPEL (which is supported as well) to orchestrate events. With BeanFlow, a fast-fail mode can be enabled to fail activities as soon as a child activity fails. [39] Another option to configure failure behavior is to use Apache Camel, which provides exception handling as well.

5.4 OpenESB

OpenESB is developed by an open source community under the coordination of Sun Microsystems [3]. OpenESB is famous for its tool support through the Netbeans IDE, and makes use of Sun's Glassfish Application Server. Open ESB is available in two distributions:

- OpenESB bundled together with the Netbeans All distribution bundle;
- Glassfish ESB, an Open ESB distribution containing the runtime and some essential components.

According to [3], the latter is well tested and more stable than the former.

5.4.1 Tool Support

OpenESB is strongly linked with an Netbeans, and has been developed in the same company. Many NetBeans extensions are available which are used extensively in OpenESB examples, such as:

- **Composite Application Service Assembly (CASA) editor**

  A graphical editor of the configuration files.

- **JBI Manager**

  An application that controls the life cycle of JBI components and service assemblies.

- **WSDL Editor**

  A wizard to create or edit WSDL files.

- **XML Schema Editor**

A wizard to create or edit XML Schemas.

Because of the graphical editor of configuration files, called *JBI Service Assemblies,* it has the best tool support of the three implementations, which is acknowledged in [36]. However, [4] states that XML and JBI knowledge is absolutely necessary to be able to manually correct the configuration files. Even though the tool support is available for NetBeans only, OpenESB can be used inside other IDEs which make use of the Ant build tool (such as Eclipse) as well.

5.4.2 Components

OpenESB's most important components can be categorized as following [31]:

- **Logic and orchestration**
  Several service engines to define business logic and processes in, for example, WS-BPEL 2.0.
- **Basic interfacing**
  Binding components for e-mail, a file system, an FTP server, HTTP, JMS, JMSJCA, WS-Notification, and a scheduler.
- **Databases and data manipulation**
  Components for database persistency and data transformations.
- **Other interfaces**
  Components for command invocation, communication with LDAP servers, a generic adapter for TCP/IP.

5.4.3 Hello World example

Creating services for OpenESB takes some practice with the provided tools, and a basic understanding of BPEL is necessary, as OpenESB requires the use of BPEL to define the processes on the bus. Processes can be defined with a visual tool, which generates the BPEL documents. The BPEL process in Listing 4, which uses the WSDL from Appendix C, defines the Hello World example for OpenESB.

```
<?xml version="1.0" encoding="UTF-8"?>
<process
    name="helloWorld"
    targetNamespace="http://esb.research.traserv.com/bpel/HelloWorld"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sxt="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/
                Trace"
    xmlns:sxed="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/
                Editor"
    xmlns:sxat="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/
                Attachment"
    xmlns:sxeh="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/
                ErrorHandling"
    xmlns:tns="http://esb.research.traserv.com/bpel/HelloWorld/helloWorld"
    xmlns:sxed2="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/
                Editor2">
    <import
          namespace="http://esb.research.traserv.com/wsdl/HelloWorld"
          location="helloWorld.wsdl"
          importType="http://schemas.xmlsoap.org/wsdl/"/>
```

```
    <partnerLinks>
        <partnerLink name="PartnerLink"
          xmlns:tns="http://esb.research.traserv.com/wsdl/HelloWorld"
          partnerLinkType="tns:helloWorld"
          myRole="helloWorldPortTypeRole"/>
    </partnerLinks>
    <variables>
        <variable name="outputVar"
          xmlns:tns="http://esb.research.traserv.com/wsdl/HelloWorld"
          messageType="tns:helloWorldOperationResponse"/>

        <variable name="inputVar"
          xmlns:tns="http://esb.research.traserv.com/wsdl/HelloWorld"
          messageType="tns:helloWorldOperationRequest"/>
    </variables>
    <sequence>
        <receive name="Receive" createInstance="yes"
          partnerLink="PartnerLink" operation="helloWorldOperation"
          xmlns:tns="http://esb.research.traserv.com/wsdl/HelloWorld"
          portType="tns:helloWorldPortType"
          variable="inputVar"/>

        <assign name="Assign">
            <copy>
                <from variable="inputVar" part="input"/>
                <to variable="outputVar" part="output"/>
            </copy>
        </assign>

        <reply name="Reply" partnerLink="PartnerLink"
          operation="helloWorldOperation"
          xmlns:tns="http://esb.research.traserv.com/wsdl/HelloWorld"
          portType="tns:helloWorldPortType"
          variable="outputVar"/>
    </sequence>
</process>
```

Listing 4: BPEL process for Hello World with OpenESB

The endpoint is described in the `partnerLink` element. The copying of the input to the output takes place in the `assign` element. The `receive` and `reply` elements connect the bus to the calling application, by referring to the general endpoint through the `partnerLink` reference.

We could have opted to use a POJO for this mapping, as for example is done in [32], and we have done for the other two ESBs. That way it is possible to keep a stricter boundary between configuration and logic, but we intended to point out the possibility to mix these two, which is only available in OpenESB.

5.4.4 Requirements compliance

OpenESB complies to the still open requirements as following:

1. **Global configuration**

   The GlassFish Admin Console is a GUI which can be used to configure all the servers in the cluster.

2. **Message speed**

According to [3], OpenESB performs better than ServiceMix, but according to a claim in [24], not as good as Mule.

3. **Fail fast adequacy**

   Just as for ServiceMix, Apache Camel can be used for process orchestration. Apache Camel allows the user to configure failure behavior.

5.5 Comparison

In this section, first the comparison results of [36] presented, and updated. Then we present the results of the comparison in [4]. Finally a choice is made based on our own requirements.

5.5.1 Comparison by Rademakers

Criterion	Mule	ServiceMix	OpenESB
Support for ESB core functionality[1]	+	+	+/-
Well-written documentation	+	+/-	+
Market visibility	++	+	+/-
Active development and support community	++	+	+/-
Flexible and easily extensible with custom logic	++	+	+/-
Support for a wide range of transport protocols and connectivity options	+	+	+/-
Integration with other open source projects	++	++	+/-
Productivity with IDE support	+	+	++

Table 3: Open source ESB comparison from [36].

[36] identified eight criteria to compare ESBs, and graded the three implementations as shown in Table 3. However, the authors note that this classification is slightly subjective, and is a snapshot in time. The increasing number of components of OpenESB have increased the level of its ESB core functionality support. Its market visibility is closing in on Apache's ServiceMix, from which automatically the supporting community is expected to increase. Also the range of supported transport protocols will be more than sufficient for our case study. The lack of integration with other open source projects is however still a drawback, for example, the lack of integration with Spring, as pointed out in [3].

[1] The ESB core functionality is according to [36]: location transparency, transport protocol conversion, transformation, routing, message enhancement, security and management.

5.5.2 Comparison by Biberger

In Table 4 the main points from the open source ESB comparison from [4] is shown. This comparison however, has a few limitations. The first is that it is based on the distributions as they are, not how they can be extended by external libraries or custom code. For example, Mule does not receive the full amount of points for the category "message forwarding", because of its rather limited support for publish/subscribe mechanisms. However, on Mule's website an example is available how ActiveMQ can be used inside a Mule configuration to have a more complete mechanism for this purpose.

Criterion	Wt.	Mule	ServiceMix	OpenESB
Message forwarding	3	96	100	100
Location-transparency	2	100	100	100
Transformation	2	100	100	100
Security	2	94	83	94
Reliability	2	83	100	76
Administration and monitoring capabilities	2	100	63	50
Support for open standards and extensibility	3	75	100	100
Handling in the context of the example scenarios	2	67	76	76
Other requirements from a business perspective	3	87	87	100
Total				
Final score		**87**[2]	**91**	**90**

Table 4: Open source ESB comparison from [4].

Another limitation of this comparison, is that only Mule's community edition is evaluated. As a consequence, the reliability score is based on an evaluation or pre-production version, and the JBI compliant version (Mule JBI) is ignored in the open standard scoring.

5.5.3 Comparison based on information broker requirements

From this chapter we can conclude that all ESBs provide the same core functionality, due to open standards and Java specifications. All three ESBs have binding components for many different protocols, where the core set of protocols remains the same. They also all provide XSLT message transformations. Mule is a little more flexible when it comes to the content of the messages, where OpenESB focuses on SOAP messages, as most of the creator's examples use BPEL. Because a typical information broker cannot enforce their information suppliers to follow such standards, this lack of other examples can complicate configuration for OpenESB.

[2] According to our calculation, this should be 89.

Requirement	Weight	Mule	ServiceMix	OpenESB
Global configuration	1	2	0	2
Message speed	4	2	0	1
Fail fast adequacy	3	2	2	2
Total				
Final score		**16**	**6**	**12**

Table 5: Open source ESB comparison based on information broker requirements

As seen in Table 5, Mule is the ESB which complies to our requirements best. The implementations have been awarded points in a similar way to the system used to compare the different architectures. The scores in the table are based on the arguments from the requirements compliance subsection of each of the introductions from this chapter.

Based on the introductions of the ESBs, and the comparisons from two referenced sources, we can conclude that Mule still is the most flexible in message formats, the best supported, and easiest to integrate with other open source projects. Setting up a Hello World example for Mule took the least effort of the three implementations, because it does not require extensive knowledge of other frameworks or standards. ServiceMix required extensive knowledge of Maven and JBI, while OpenESB required some knowledge of BPEL. The strict separation of configuration and logic in Mule was another reason to choose Mule for our proof of concept.

6. Proof of Concept

To evaluate the proposed architecture, this chapter describes how Mule can be used to integrate an information broker with its information suppliers. For a basic introduction to Mule we recommend the first six chapters of [12]. Section 6.1 describes the problem situation, Section 6.2 introduces briefly the basic concepts and terminology of Mule, Section 6.3 shows how the applications of the information broker and the third parties can be mapped to the building blocks of Mule, Section 6.4 explains the testing environment, and Section 6.5 concludes this chapter by assessing the quality of this solution.

6.1 Problem situation

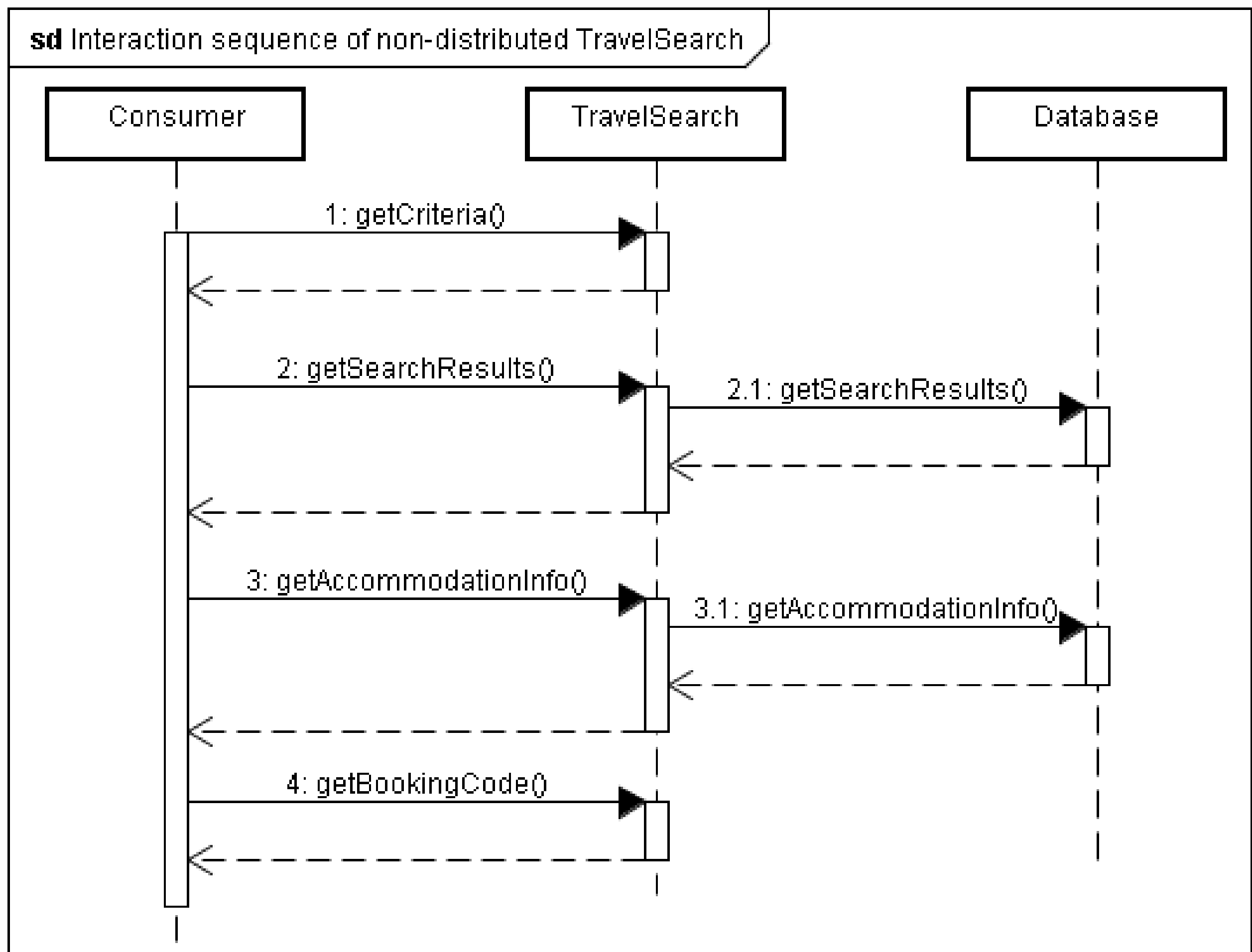


Figure 23: Interaction sequence of non-distributed TravelSearch

The current situation at our case study information broker is the following: a Java implementation of the TravelSearch interface receives search requests from the consumers for product information. This implementation queries a database, based on statically defined criteria. This implementation is made public to the consumers as a web service. TravelSearch offers a drill-down search functionality, which consists of four main steps:

1. Retrieve possible search criteria

2. Retrieve a collection of search results based on one or more of these search criteria
3. Retrieve specific information of one of these search results
4. Retrieve a booking code for this specific result, based on specific conditions

Figure 23 shows how this works in practice. The Distributed TravelSearch (DTS) which we have implemented with Mule follows this sequence diagram too, but connects to third party web services, instead of connecting to the database.

Creating a web service which focuses on the first step would require a commonality analysis of the underlying external web services. As this commonality analysis is most likely to be done statically, this would form an integration challenge, however not one which can be solved with an ESB. The fourth step would have been very implementation-specific, as this booking code is the input of another web service of TSi Solutions, rather than based on information of external services, and therefore not the challenge we were looking for either.

From the second and the third step we have chosen the challenge of the third step, as this step allowed us to focus on configuring Mule to use an external web service, instead of drifting of to response aggregation difficulties, such as identification of identical products which are available through more than one external service. Transformation of the results of the external services was an unavoidable side-effect which we would have had to face with both options. With the basis that we have laid by configuring Mule for the third step, attempting to use several web services simultaneously would be an interesting next step in this research. The aggregator class could be extended for this purpose.

6.2 Basics of Mule

Mule's terminology is based on the terminology used in [17]. Every application in Mule is a *service*. A service is built up of several building blocks, as shown in Figure 24.

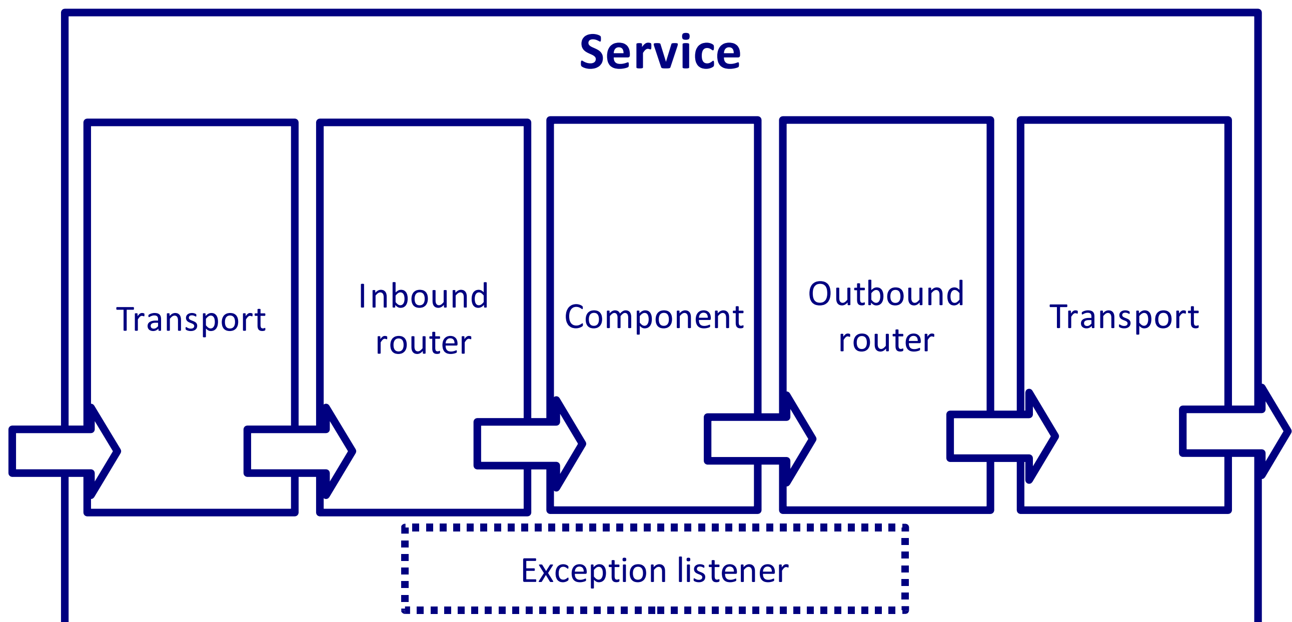


Figure 24: Outline of a Mule service [12]

The arrows on the left and right are *inbound*- and *outbound-endpoints,* respectively. They indicate where data enters the bus and where it exits the bus. Endpoints always have an address in the form of a URI, although these are almost never found explicitly in the configuration files, as Mule creates them based on the attributes of the endpoint configuration and the namespace of the element. A useful endpoint for testing purposes is the console.

A *transport* can be of many different types, such as HTTP, SMTP, IMAP, and FTP. Transports are technologies of which only the bus should be aware of, not the applications.

Routers define where a 'message' (a file, an email, an HTTP request, etc.) should go to, or if it should even go anywhere, based on its content. Examples are the *pass-through-router*, which lets everything go through, or the *chaining-router*, which uses the output of one endpoint as the input for the next one. *Filters* can be used to block messages which do not comply to some requirements, like message structure.

Components are the actual processors of the services which are accessible through Mule. They are applications, such as, for example, web service implementations.

Figure 25 shows how these building blocks can work together in order to support service execution. *Transformers*, depicted as small circles with a 'T' in them, are capable of translating messages, for example, by using an XSLT transformation. The mentioned protocols and component type in Figure 25 are examples, and the outbound endpoint, transformers and the component are optional. Transformation may also be done through a chain of transformers.

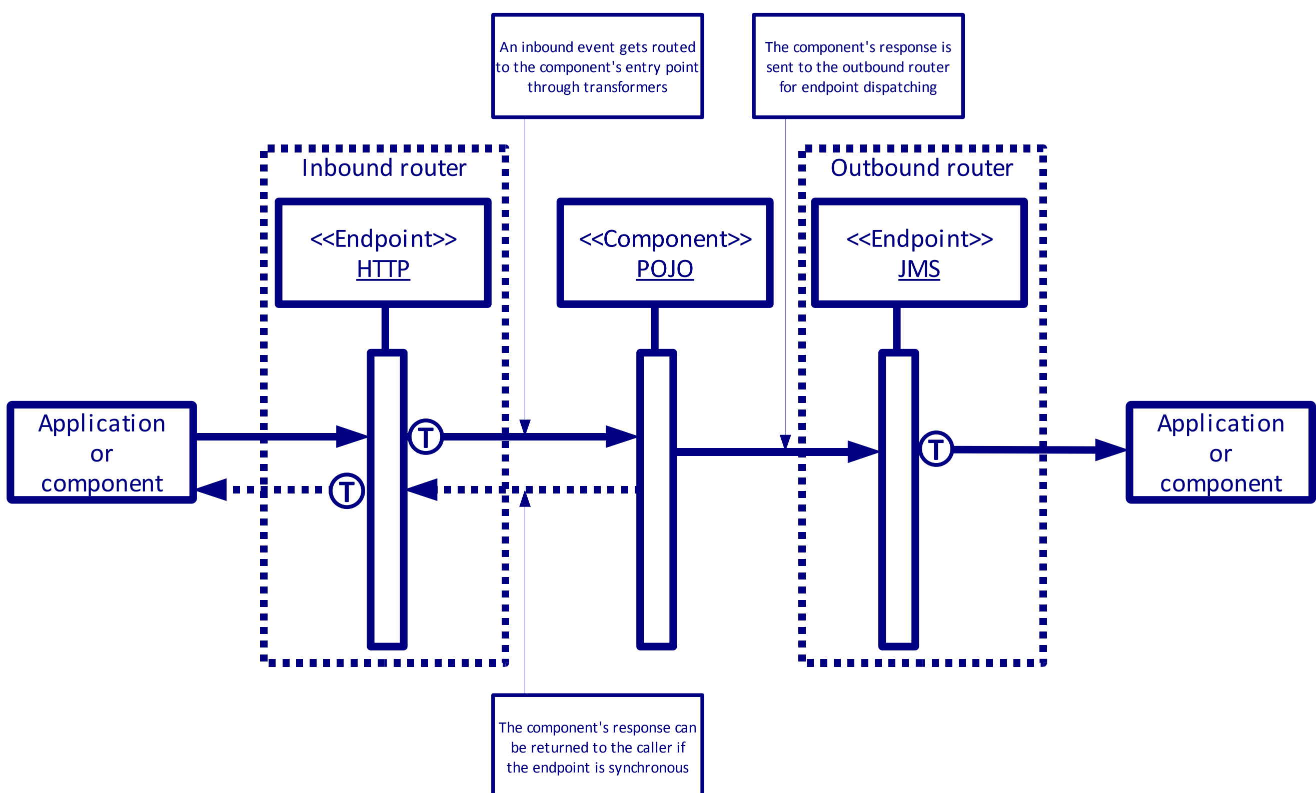


Figure 25: Mule sequence diagram [12]

The two previous figures can be related to each other in the way displayed in Figure 26. The exception handler is left away in this figure, since the sequence diagram demonstrates the

intended flow of information. The inbound transport is HTTP, the component is implemented as a POJO, and the outbound transport is Java Message Service (JMS).

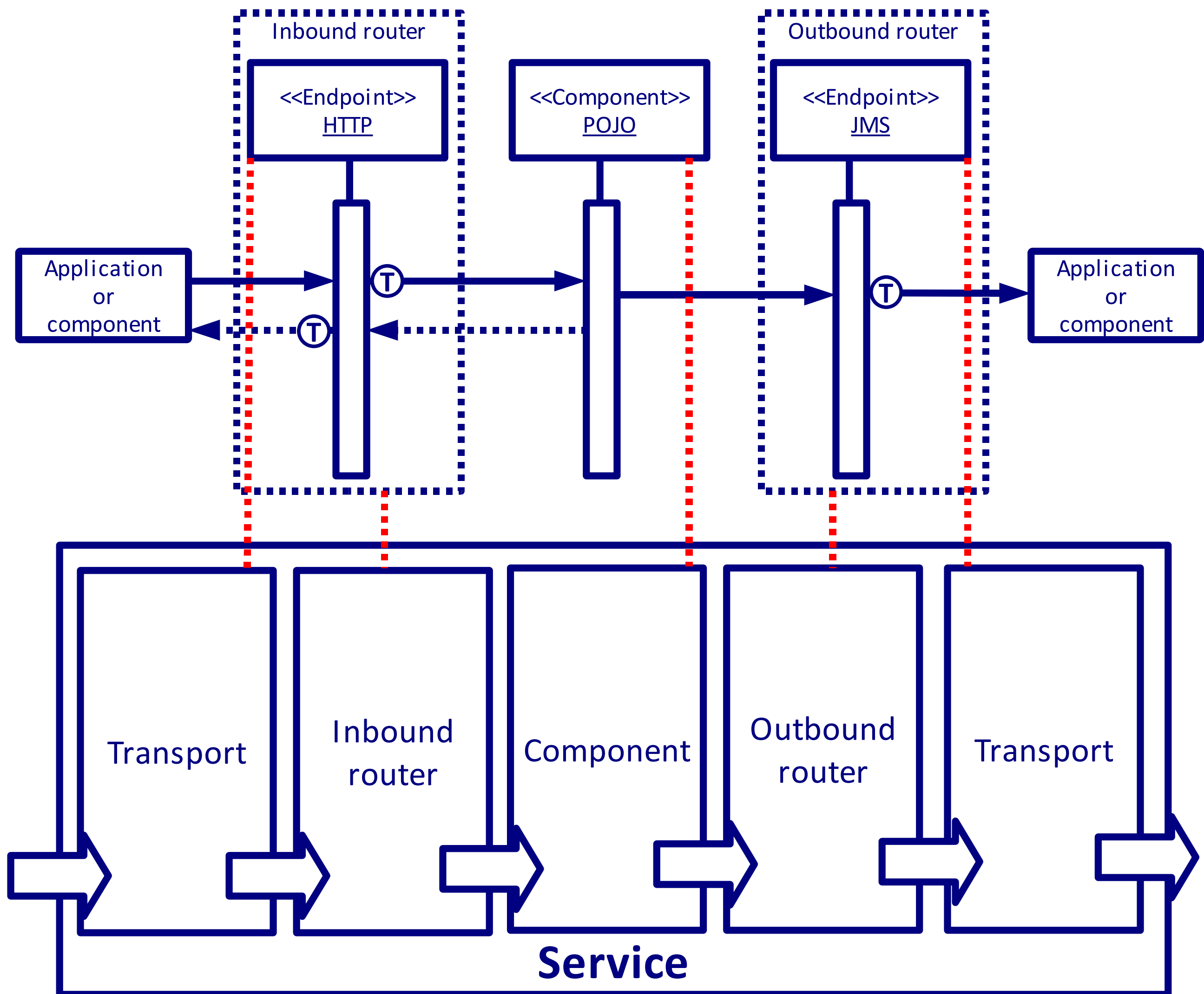


Figure 26: Mapping of sequence diagram onto service description

6.3 Configuring Mule for an information broker

For an information broker the applications which they offer or are connected to can be split up into three groups: producer applications, consumer applications, and broker applications. The role of the latter is to connect the former two.

To demonstrate the possibility of using Mule for an information broker we needed an information consumer, a producer, and a Mule configuration that connects those two. For the consumer we have chosen a SOAP client (e.g., soapUI). For the source we have created a mock up implementation of a real search engine for hotels and apartments (amongst others).

By using an existing WSDL document from the case study company, and a mock up for a web service of a producer that is implemented in practice, we have been able to demonstrate how mismatches between the two interfaces can be solved. For this example, however, we have not attempted to provide an implementation for all operations defined in the WSDL file, nor the mock up implementation, but we have chosen one representative operation.

In the Mule configuration, the inbound endpoint (which we have chosen to be a SOAP web service) is the web service that the consumer needs to contact. The web services of the producers are assigned to the outbound endpoints. Since the producers do not communicate with the same message structure, we need to transform the SOAP messages to the respective requests and the corresponding responses back to SOAP messages. Therefore for the mock up producer, we had to create two main transformations. For each newly added producer, two more transformations would need to be created. Such a mapping of variables is necessary for other integration solutions as well, and even for the information broker approach with a database as information source, this mapping needs to be carried out for the database update application.

Figure 27 shows how Mule can implement the information broker scenario. The names of the building blocks match the names of the XML elements in the configuration file in Listing 5. As depicted in Figure 27, the consumer calls the SOAP operation. This SOAP request is transformed to transformable XML, since the interface of the SOAP web service is very generic, with a lot of freedom for the values of fields and complex combinations of these values. This XML message is then passed on by the `WSProxyService`, and routed to a message queue (`dtsRequests`) to which the wrappers for the external services are subscribed.

The responses from the external services come back on another message queue (`dtsResponses`), where they are collected and merged into one response in the `DTSResponseAggregator`. This response is transformed one more time to a SOAP response message.

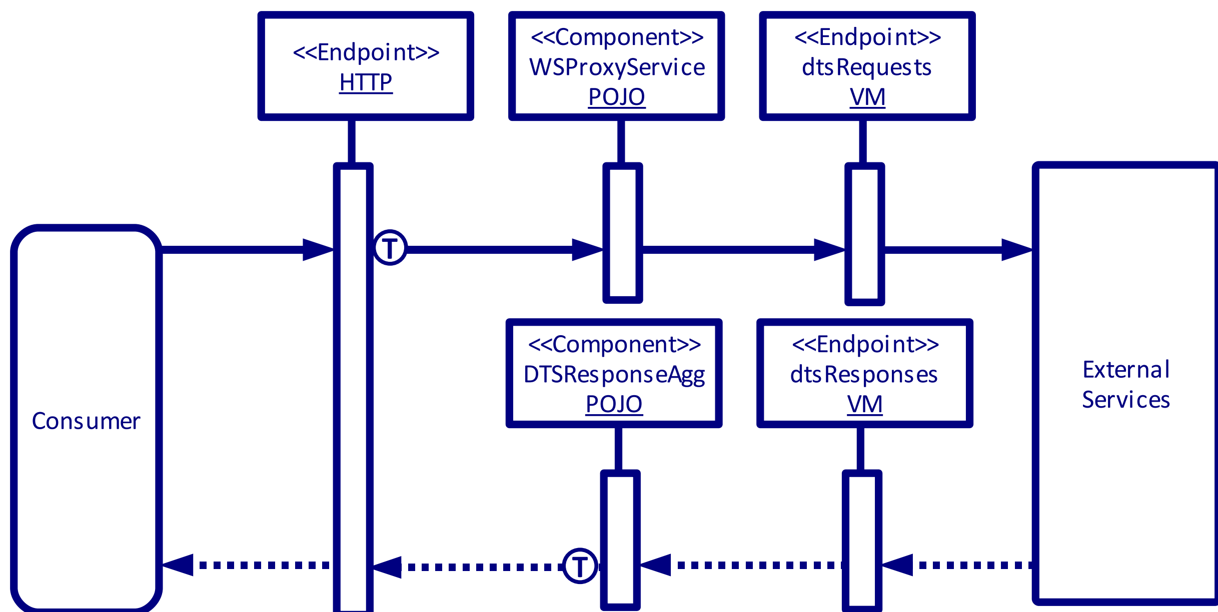


Figure 27: Main service in the Mule configuration (`dtsService`)

Figure 28 provides us with a closer look at the right-most side of Figure 27 (the *external services* block), and shows how the *virtual machine message* is handled by the service definitions for external services. Virtual machine messages are Mule's Java implementation of Message-Oriented Middleware. The message stays inside the Java Virtual Machine and is passed on to the services subscribed to the topic the message is published on.

As seen in Figure 28, the incoming request is transformed from the canonical XML to the specific

XML for this provider (as the mock up implementation expects an XML message over HTTP). The response from the external web service is transformed back to the canonical XML structure, and then passed on to the response queue. Figure 28 does not show that filtering takes place to determine whether a message is intended for this service.

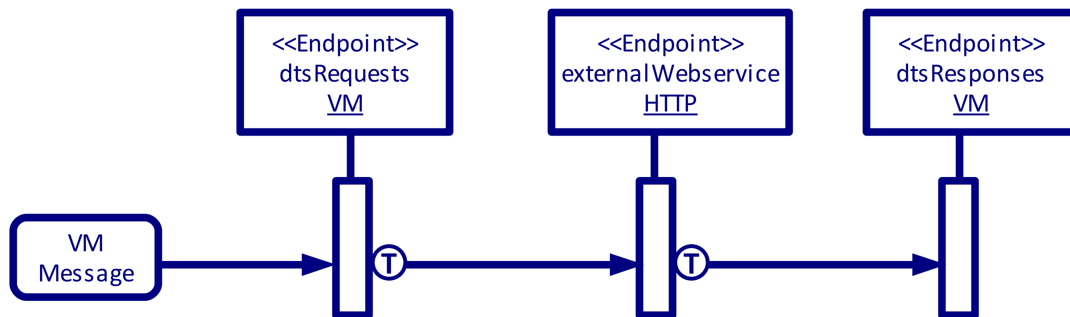


Figure 28: Specific provider configuration (`externalServiceService`)

The configuration used to implement the scenario above in Mule is shown in Listing 5. For brevity, all namespaces and error handling code have been omitted. The abbreviation DTS stands for Distributed Travel Search, which is the name of the case study project. The configuration consists of the following items:

- **Spring Beans**

  As the defined web service does not actually do anything except for collecting and transforming data, no service class needs to be implemented. The WSProxyService bean takes the role of the service class, and passes the message on to the outbound endpoints.

- **Transformers**
  We defined four transformers: two to translate the SOAP messages to transformable XML and back, and two to transform this XML to the message structure of the external web service.

  For every new external service two more transformations would be necessary (one for requests and one for responses). The decision for the type of transformation (through an XSLT, a Java class or any other option) would have to be made individually for each new service.

- **Endpoints**
  Currently, there are four endpoints:
  - The inbound endpoint of the main service;
  - The outbound endpoint of the external service;
  - The message queue for requests to external services;
  - The message queue for responses from the external services.

  Any new external service would introduce one more endpoint, while the response and request endpoints can be reused.

- **Model**

  A 'model' in Mule contains one or more services. In our case, the model consists of two

services: the main service and the external service. Any new external service would imply one extra service inside this model. This new service would then be added inside the multicasting-router on the outbound endpoint of the main service.

- **Main service**

  The main service (dtsService) defines the inbound endpoint for requests, the WSProxyService to catch the SOAP requests, and broadcasts the requests to all external services. The async-reply element is used to collect their responses. In case a new service is added, an extra outbound endpoint needs to be added to the multicasting-router.

- **External service**

  The external service defines the specific third-party service. The inbound endpoints define to which queues the external service is listening, and the outbound endpoint defines where the service is located. The chaining router routes the response of the service to the response queue which collects these. For each new external service, a copy of this configuration should be made and tailored, but no changes shall be made to this service.

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2">

  <!-- SPRING BEANS -->
    <spring:bean name="WSProxyService" class="org.mule.transport.soap.WSProxyService">
        <spring:property name="wsdlFile" value="dts.wsdl"/>
    </spring:bean>

  <!-- TRANSFORMERS -->
  <custom-transformer name="dtsSoapRequest2xml" class="com.traserv.research.esb.SoapToXML"/>
  <custom-transformer name="xml2dtsSoapResponse" class="com.traserv.research.esb.XMLToSoap"/>
  <xm:xslt-transformer name="dts2externalService"
    xsl-file="transformations/dts/externalService/dts2externalService.xsl"
    ignoreBadInput="false" returnClass="java.lang.String"/>
  <xm:xslt-transformer name="externalService2dts"
    xsl-file="transformations/dts/externalService/externalService2dts.xsl"
    ignoreBadInput="false" returnClass="java.lang.String"/>

  <!-- ENDPOINTS -->

  <!-- This is the DTS endpoint as it is seen by consumers of this service -->
  <http:endpoint name="dts" address="http://localhost:8081/dts" synchronous="true">
    <transformers>
      <transformer ref="dtsSoapRequest2xml"/>
    </transformers>
    <response-transformers>
      <transformer ref="xml2dtsSoapResponse"/>
    </response-transformers>
  </http:endpoint>

  <!-- Connects to the external service, which is running outside Mule -->
  <http:endpoint name="externalService"
                 address="http://localhost:8080/externalService/relay"
                 synchronous="true"/>

  <!-- Collects the responses of the individual external web services, in order
       to make one response out of these -->
  <vm:endpoint name="dtsResponses" address="vm://dts.responses"/>

  <!-- Enqueues the messages for external services -->
  <vm:endpoint name="dtsRequests" address="vm://dts.requests" />


  <!-- MODEL -->
  <model name="dtsModel">
        <service name="dtsService">
          <inbound>
```

```
        <inbound-endpoint ref="dts"/>
      </inbound>

      <component>
        <spring-object bean="WSProxyService" />
      </component>

      <outbound>
        <multicasting-router>
          <vm:outbound-endpoint ref="dtsRequests" synchronous="true"/>
        </multicasting-router>
       </outbound>

       <async-reply timeout="2000">
          <vm:inbound-endpoint ref="vm://dtsResponses" synchronous="true" />
          <custom-async-reply-router class="com.traserv.research.esb.DTSResponseAggregator" />
        </async-reply>
    </service>

    <service name="externalServiceService">
      <inbound>
        <vm:inbound-endpoint ref="dtsRequests" synchronous="true">
          <transformers>
            <transformer ref="dts2externalService"/>
          </transformers>
        </vm:inbound-endpoint>
      </inbound>

      <outbound>
        <chaining-router>
          <outbound-endpoint ref="externalService" synchronous="true"/>
          <outbound-endpoint ref="dtsResponses">
            <transformer ref="externalService2dts"/>
          </outbound-endpoint>
        </chaining-router>
      </outbound>
    </service>
  </model>
</mule>
```

Listing 5: The Mule configuration for our prototype

6.4 Testing environment

In order to clarify how the discussed configuration has been tested, this section discusses how the main service (the Distributed Travel Search) and the external web service (the mock up of an existing producer service) have been simulated.

6.4.1 Distributed Travel Search

The classes used for the DTS are:

- **com.traserv.research.esb.DistributedTravelSearch**

  This class transforms the very generic WSDL definition classes to XSLT suitable XML.

- **com.traserv.research.esb.DTSResponseAggregator**

  This class generates Java objects for the WSDL endpoint out of the responses of the external web services.

6.4.2 External web service

The external web service is a mock up for an existing travel search web service. This mock up returns a valid response for the given request type. This functionality is provided through a servlet,

which first extracts the request type, then loads the corresponding response from an XML file, validates this through the Castor framework [8], and then responds this XML. The servlet runs on an Apache Tomcat server [43]. The service supports 48 different requests, of which we have only used the one that returns information for a specific accommodation.

The mock up is shielded in Mule by two main transformations. Without going too much into detail for these transformations, we present the top level transformation for requests and responses. These transformations refer to other XSLTs which have specifically been written for each request or response type.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:output
            method              = "xml"
            encoding            = "utf-8"
            omit-xml-declaration ="no"
            standalone          = "no"
            indent              = "yes"
      />

  <xsl:include href="transformations/dts/externalService/requests/header.xsl"/>
  <xsl:include
    href="transformations/dts/externalService/requests/SearchItemRequest.xsl"/>

  <xsl:template match="/">
    <xsl:apply-templates select="request"/>
  </xsl:template>

  <xsl:template match="request">
    <Request>
      <xsl:attribute name="TimeStamp">
        <xsl:value-of select="@time"/>
      </xsl:attribute>

      <xsl:attribute name="RequestReference">
        <xsl:value-of select="@referenceCode"/>
      </xsl:attribute>

      <xsl:call-template name="header"/>
        <RequestDetails>
        <!-- one of the templates below will be picked up, based on the type of
             request -->
          <xsl:apply-templates
        select="getAccommodationInfo[@criteriumName = 'specificAccomodation']"/
>
        </RequestDetails>
      </Request>
    </xsl:template>
</xsl:stylesheet>
```

Listing 6: Request transformations

As shown in Listing 6, which describes the request transformations, a new file can be included for each supported request, to keep a clear hierarchy in the transformations. For each newly supported request type, a new line is added inside the `RequestDetails` tag, with an XPath expression to select the appropriate template. The generic name of the only supported call so far

is getAccommodationInfo (the input of the request transformation is the canonical model), and an attribute for the criterion name has been added.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output
          method               = "xml"
          encoding             = "utf-8"
          omit-xml-declaration ="no"
          standalone           = "no"
          indent               = "yes"
  />

  <!-- Response includes -->
  <xsl:include
   href="transformations/dts/externalService/responses/SearchItemResponse.xsl"/
>

  <xsl:template match="/">
    <response tourOperator="externalService">
     <!-- one of the templates below will be picked up, based on the
           type of response -->
     <xsl:apply-templates select="Response/ResponseDetails/SearchItemResponse"/
>
    </response>
  </xsl:template>
</xsl:stylesheet>
```

Listing 7: Response transformations

Listing 7 shows the transformation of responses. The structure of a response for this external service is a Response element, with a child element ResponseDetails, which in its turn has a child element with the name of the type of response. Again, for each supported response a new file is included, and another line to call the corresponding template is added, this time inside the response tags, again to keep a hierarchical structure for the transformation files.

6.5 Conclusion

In this chapter, we have shown how Mule can be configured, supported by Java code and XSLT transformations, for the purpose we intended. We have also shown how the configuration can be extended on the long run to support more services or operations. Though we have not tested our configuration for all possible operations, how to use Mule's security layer or retry policies, we have configured Mule to carry out all the functional requirements.

To test Mule's response time under increased load, we have performed a load test, from which the Java code can be found in Appendix D. As the load on the server increases, the response time increases dramatically, as can be seen in Figure 29. Transforming all the responses simultaneously consumes a substantial amount of memory and processing power, leading to unacceptable response times for more than 100 simultaneous requests per server under the given circumstances. In our simulation, no new calls to the server have been made after these concurrent requests. From this test, we can conclude that Mule does not scale under these circumstances. However, the circumstances, where external services are running on the same physical server as the ESB, do not conform to the intended use of the ESB.

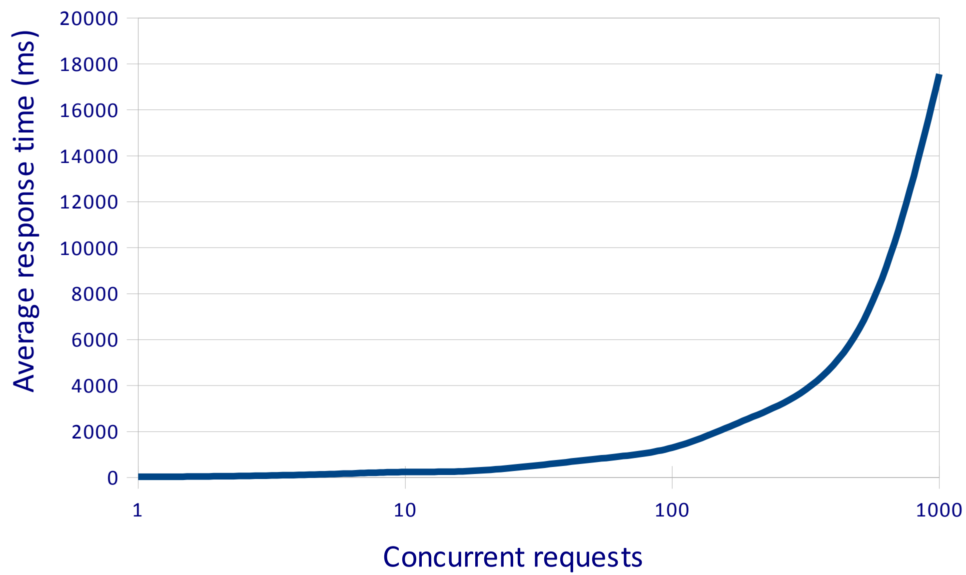


Figure 29: Load test results

This scalability issue can be solved by assigning more memory to Mule (as this was limited by running the load test, the external service and Mule on the same machine), using multiple Mule instances in a cluster [12], and possibly by performing optimizations on the transformations, for example by using templates with placeholders, rather than building up the XML for each request and response. With the latter optimization option, it might even be possible to eliminate the intermediate transformation step.

7. Final Remarks

This chapter presents the final remarks of this research, by presenting conclusions and recommendations for further research. This chapter is further structured as follows: section 7.1 presents the conclusions and the contributions of this thesis, section 7.2 identifies which issues are still open.

7.1 Conclusions

The main objective of this research was to *determine the most suitable architecture and technology to realize distributed access on real-time to information on products*.

As a step towards this architecture, we have discussed four integration technologies. After eliminating the point-to-point integration, based on the maintenance difficulties, we have presented seven different possibilities, making use of current technologies, such as EAI, EMB and ESB.

Based on requirements identified through an information broker analysis, a questionnaire and extensive discussions, we have proposed an ESB architecture to integrate all the information broker's application with external applications. The message transformation possibilities, ease to add external applications, and current market tendencies made ESB stand out as best integration technology. The ability to route requests from consumers directly to producers made us propose to use the ESB company-wide, and as the entry point for requests by information consumers. This architecture enables developers to focus on the transformations of the content of the offered services and consumed external services, rather than on protocol mismatches.

We have introduced three of the most popular ESB implementations by analyzing their tool support, existing components and through a Hello World example. Based on the information broker's requirements, and two existing comparisons from literature, we have compared the three implementations and found Mule the one which suites the application in the information broker domain best. This result was reinforced by Mule's flexibility with regard to the content of the messages, its supporting community, and its integration with other open source projects.

We have validated the possibility to use Mule for an information broker through a prototype. With the use of message queues, requests can be distributed between multiple external services, and their responses can be collected. Transformers enabled us to overcome the problem of all external services having different message structures. In the validation of our configuration, we have shown the importance of light-weight transformations between the different models, to be able to cope with heavy loads on the server.

7.2 Future research

A point of research which still has to be done before the proposed architecture can be brought into operation is to validate our architecture while consuming more than one external service. Interesting integration challenges, such as identification of identical products which are offered by several external parties, will need to be overcome. Then the performance of the solution will also need to be tested with services running on different machines, access of the services over network connections, rather than at a local machine, and under a realistic load.

The most interesting field of research which we have only touched briefly upon, is the area of transformation optimization. The XSLT definitions which we have written have a high maintainability, but have proven to consume a lot of processing power and memory. Research could be done on how to write more efficient transformations, but other possibilities could be to build tools that parse the XSLT definition and generate specific and more efficient code. Another optimization could be obtained by developing tools that are capable of merging a chain of transformations into one transformation.

A last point of future research is on the comparison of ESBs from a performance point-of-view. In the comparison in [36] performance has not been included explicitly. The comparison in [3] is based on two ESBs running on one local machine and evaluates message exchange speed only. The comparison in [4] only evaluates the reliability of the ESBs and not the message speed. None of the comparisons perform a load test of the system, nor did they use clustering. For a complete ESB comparison based on performance, many resources will be needed, such as:

- multiple ESB servers in a cluster, to analyze the scaling capabilities;
- a complex problem scenario, which touches upon many components, updates configurations, and adds and removes service locations frequently;
- experienced users (or even the developers) of several ESB implementations to use the ESBs to their full capacity;
- complete licenses for all ESBs, for a fair comparison.

References

[1] Apache ActiveMQ.
Apache ActiveMQ – Index.
activemq.apache.org

[2] Ferenc Bator.
Enterprise Application Integration.
www.devmatic-it.com/devmatic-it/en/article/eai

[3] Thomas Bayer.
OpenESB and ServiceMix in Comparison
www.predic8.com/openesb-servicemix-comparison.htm

[4] Ulrich Biberger.
Open-Source Enterprise Service Bus Lösungen im Vergleich. (Open source ESB solutions compared)
Grin, 2009.

[5] BizTalk website.
Microsoft BizTalk website.
www.microsoft.com/biztalk/en/us/default.aspx

[6] Herbert Burbiel.
SOA & Webservices in der Praxis. (SOA & Webservices in Practice)
Franzis Verlag, 2007.

[7] Vincenzo Caselli, Binildas A. Christudas, and Malhar Barai.
SOA with Service Component Architecture and Enterprise Service Bus.
www.packtpub.com/article/service-component-architecture-message-oriented-middleware-enterprise-service-bus

[8] Castor website.
The Castor Project.
www.castor.org/

[9] Dave Chappell.
Enterprise Service Bus.
O'Reilly, Sebastopol (CA), 2004.

[10] Binildas A. Christudas.
Service Oriented Java Business Integration.
Packt Publishing, Birmingham, 2008.

[11] Delta Airlines website.
Delta through the decades.
www.delta.com/about_delta/corporate_information/delta_stats_facts/delta_through_decades/index.jsp

[12] David Dossot and John D'Emic.
Mule in Action.
Manning Publications Co., Greenwich, 2009.

[13] Naveen Erasala, David C. Yen, T.M. Rajkumar.
Enterprise Application Integration in the electronic commerce world.
tinyurl.com/ygpetj4

[14] Richard Fikes, Robert Engelmore, Adam Farquhar, Wanda Pratt.
Network-Based Information Brokers.
Knowledge System Laboratory, Stanford University, 1995.

[15] Julie Gable.
Enterprise Application Integration.
Information Management Journal, Mar/Apr 2002.

[16] Hermann and Dieter Kessler.
XML Signatures in an Enterprise Service Bus Environment.
Springer Link, 2005.

[17] Gregor Hohpe and Bobby Woolf.
Enterprise Integration Patterns.
Addison-Wesley Professional, 2003

[18] JAX-WS website.
jax-ws: JAX-WS Reference Implementation
jax-ws.dev.java.net

[19] Java Community Process.
Java™ Business Integration (JBI) 1.0.
Sun Microsystems, Inc., 2005.

[20] Jitterbit website.
Jitterbit Open Source Integration - Business Integration Made Easy.
www.jitterbit.com

[21] Frank Kappe.
A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems.
J-UCS, 1995.

[22] Martin Keen et al.
Patterns: Implementing an SOA Using an Enterprise Service Bus.
IBM Redbooks, 2004.

[23] Martin Keen et al.
Patterns: SOA with an Enterprise Service Bus.
IBM Redbooks, 2005.

[24] Paul Krill.
MuleSource readies open source SOA governance.
www.infoworld.com/t/platforms/mulesource-readies-open-source-soa-governance-517

[25] P. de Leusse, P. Periorellis, and P. Watson.
Enterprise Service Bus: An overview.
University of Newcastle, 2007.

[26] David S. Linthicum.
Enterprise Application Integration.
Addison-Wesley, 1999.

[27] Alastair M. Morrison, Su Jing, Joseph T. O'Leary, and Liping A. Cai.
Predicting Usage of the Internet for Travel Bookings: An Exploratory Study
Cognizant, 2001.

[28] Mule website.
Home - Mulesoft Community - mulesoft.org
www.mulesoft.org

[29] Tim Oates, M.V. Nagendra Prasad, Victor R. Lesser.
Cooperative Information Gathering: A Distributed Problem Solving Approach.
IEE Proceedings-Software Engineering, 1997.

[30 Openadaptor website.
Welcome to openadaptor.org.
https://www.openadaptor.org

[31] OpenESB website.
OpenESB, The Open Source ESB for SOA & Integration - Home.
open-esb.dev.java.net

[32] OpenESB website.
OpenESB POJO Tutorial Echo Service.
wiki.open-esb.java.net/Wiki.jsp?page=POJOTutorialEchoService

[33] Open Message Queue.
mq: Open Message Queue : Open Source Java Message Service (JMS).
mq.dev.java.net

[34] Michael P. Papazoglou.
Web Services: Principles and Technology.
Pearson - Prentice Hall, 2007.

[35] Michael P. Papazoglou, Paolo Traverso, Schraham Dustdar, and Frank Leymann.
Service-Oriented Computing: State of the Art and Research Challenges.
Computer, November 2007.

[36] Tijs Rademakers and Jos Derksen.
Open Source ESBs in Action.
Manning Publications Co., Greenwich, 2008.

[37] Sandesha website.
Sandesha - Apache Sandesha.
ws.apache.org/sandesha/

[38] Roy W. Schulte.
Predicts 2003: Enterprise Service Buses Emerge.
Gartner, 2002.

[39] Apache ServiceMix website.
Apache ServiceMix, the Agile Open Source ESB -- Home.
servicemix.apache.org

[40] Si-qing Liu.
A Theoretic Discussion of Tourism E-commerce.
ACM, 2005.

[41] SRI International.
Global Impacts of FedEx in the New Economy.
www.sri.com/policy/csted/reports/economics/fedex/

[42] Ron Ten-Hove, Peter Walker, et al.
Java Business Integration 1.0 Final Release.
jcp.org/aboutJava/communityprocess/final/jsr208/index.html

[43] Apache Tomcat website
Apache Tomcat - Welcome!
tomcat.apache.org

[44] Sten A. Thore A. and Fedele Iannone.
The Hub-and-Spoke Model - A Tutorial.
ssrn.com/abstract=950753

[45] Tsi website.
TSi Solutions.
www.tsi-solutions.nl

[46] Mark Turner, Fujun Zhu, Ioannis Kotsiopoulos, Michelle Russell, David Budgen, Keith Bennett, Pearl Brereton, John Keane, Paul Layzell and Michael Rigby.
Using Web Service Technologies to create an Information Broker: An Experience Report.
'04, IEEE, 2004.

[47] Arno van der Laan.
Ontwerpen van een Enterprise Service Bus. (Design of an ESB).
Java Magazine 3, October 2006.

[48] IBM WebSphere MQ.
IBM - WebSphere MQ – Software.
www-01.ibm.com/software/integration/wmq/

[49] Wikipedia.
Mule (software).
en.wikipedia.org/wiki/Mule_%28software%29

[50] Gulnoza Ziyaeva, Eunmi Choi, and Dugki Min.
Content-Based Intelligent Routing and Message Processing in Enterprise Service Bus.
International Conference on Convergence and Hybrid Information Technolog, 2008.

Appendices

Appendix A. Requirements Questionnaire

This appendix contains the questionnaire held amongst CTO, system architects, and developers of the case study information broker. The *italic* text is the text of the questionnaire, the bullet points form a summary of the responses.

The following questions are about a possible new integration solution for an information broker. This integration solution can be seen as a black box between the consumers on one hand and the providers of the offered products on the other side, directing requests from the consumers to the applicable web service of the tour operators.

An information broker is defined as "a company which provides services to other companies to search product information and purchase these products". The goal of the questionnaire is to find out what the requirements of such an integration solution are.

1. *Who are the stakeholders of such an integration solution? Name as many as you can think of.*
    - Consumers:
        - Travel agencies;
        - Website developers;
        - Third-party website developers (who are only responsible for creating the website, but do not operate it themselves).
    - Information broker:
        - System architects;
        - Developers;
        - System administrators;
        - Project leaders.
    - Producers:
        - Tour operators;
        - Insurance companies;
        - Payment service providers;
        - Car rental providers;
        - Transportation companies (such as airline, railway and bus companies).

2. *For each of the stakeholders from question 1, please specify at a high level what actions they take which would be relevant for an integration solution?*
   For example: "search in the catalog", "turn the offers on or off for a specific producer".

- Consumers:
    - Search;
    - Book;
    - Cancel;
    - Purchase insurances;
    - Configure which tour operator's products to offer;
    - Create top 10's;
    - View reports/statistics;
    - Check quality of the catalog (reporting);
    - Add customer reviews;
    - Handle payment.
- Information broker:
    - Enrich information;
    - Direct purchase requests;
    - Direct search requests.
- Producer
    - Configure which websites are allowed to show their information.

3. *For such an integration solution, which transports (such as HTTP) and protocols (such as SOAP) should at least be supported?*

    - HTTP;
    - HTTPS;
    - FTP;
    - SOAP;
    - XML-RPC;
    - REST;
    - Java serialization;
    - Flexibility in use of protocols (not a protocol but a common remark).

4. *Which properties which you would like to see in such an integration solution?*

    - Speed;
    - Security/central authentication mechanism;
    - Extendability;

- Simplicity;
- Independence of implementation language;
- Reliable Messaging;
- Open source;
- Removal of doubles;
- Scalability;
- High availability;
- Statistics.

5. *Which properties do you consider desirable for other stakeholders?*

    - Security;
    - No change necessary for implemented services;
    - Fail fast;
    - Transparency;
    - Flexibility in application;
    - Possibility to combine products;
    - Reports and statistics;
    - Single sign-on;
    - Quality assurance of catalog.

Appendix B. Hello World Request and Response

```
<soapenv:Envelope xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:hel="http://esb.research.traserv.com/wsdl/HelloWorld/helloWorld">
  <soapenv:Body>
    <hel:helloWorldOperation>
      <input>Hello World</input>
    </hel:helloWorldOperation>
  </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <m:helloWorldOperationResponse
         xmlns:m="http://esb.research.traserv.com/wsdl/HelloWorld/helloWorld">
      <output xmlns="">Hello World</output>
    </m:helloWorldOperationResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Appendix C. WSDL document for ESB Comparison

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="helloWorld"
    targetNamespace="http://esb.research.traserv.com/wsdl/HelloWorld/helloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://esb.research.traserv.com/wsdl/HelloWorld/helloWorld"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

    <types/>

    <message name="helloWorldOperationRequest">
        <part name="input" type="xsd:string"/>
    </message>

    <message name="helloWorldOperationResponse">
        <part name="output" type="xsd:string"/>
    </message>

    <portType name="helloWorldPortType">
        <operation name="helloWorldOperation">
            <input name="input" message="tns:helloWorldOperationRequest"/>
            <output name="output" message="tns:helloWorldOperationResponse"/>
        </operation>
    </portType>

    <binding name="helloWorldBinding" type="tns:helloWorldPortType">
        <soap:binding style="rpc"
                      transport="http://schemas.xmlsoap.org/soap/http"/>

        <operation name="helloWorldOperation">
            <soap:operation/>
            <input name="input">
                <soap:body use="literal"
                           namespace="http://esb.research.traserv.com/wsdl/Hello
                                      World/helloWorld"/>
            </input>
            <output name="output">
                <soap:body use="literal"
                           namespace="http://esb.research.traserv.com/wsdl/Hello
                                      World/helloWorld"/>
            </output>
        </operation>
    </binding>

    <service name="helloWorldService">
        <port name="helloWorldPort" binding="tns:helloWorldBinding">
            <soap:address
             location="http://localhost:8080/helloWorldService/helloWorldPort"/>
        </port>
    </service>
    <plnk:partnerLinkType name="helloWorld">
        <plnk:role name="helloWorldPortTypeRole"
                   portType="tns:helloWorldPortType"/>
    </plnk:partnerLinkType>
</definitions>
```

Appendix D. Load test for the Mule configuration

```
package com.traserv.research.esb.mule.util;

import org.mule.api.MuleException;
import org.mule.module.client.MuleClient;

/**
 * This class tests Mule how it handles heavy traffic.
 */
public class LoadTester {
	public static int threadsCreated = 0;

	public final static String SOAP_REQUEST = "<soapenv:Envelope .. </soapenv:Envelope>";

	public final static String URL = "http://localhost:8081/dts";

	private MuleClient client;

	/**
	 * in Nanoseconds
	 */
	private long totalTime = 0;

	private int finishedTests = 0;

	/**
	 * in Nanoseconds
	 */
	private long maxResponseTime = 0;

	public static void main(String[] args) throws Exception {
		LoadTester loadTester = new LoadTester();
		loadTester.runTests(1000);
	}

	public LoadTester() {
		try {
			client = new MuleClient();
			client.send(URL, SOAP_REQUEST, null);
			// The first send affects results, as the MuleClient is
			// initialized (around 10x slower than other results).
		} catch (MuleException e) {
			e.printStackTrace();
		}
	}

	public void runTests(int nrOfTests) {
		long start = System.nanoTime();

		for (int i = 0; i < nrOfTests; i++) {
		     new Test(); // private class on next page
		}

		while (finishedTests < nrOfTests) {
		}

		long end = System.nanoTime();

		System.out.println("Total time: " + ((end - start)/1000000) +
		                      " ms for " + nrOfTests + " tests (in parallel).");
		System.out.println("Average time per test: " +
		                      (totalTime/(1000000 * nrOfTests)) + " ms\n");
		System.out.println("Max response time: " + (maxResponseTime/1000000)
			+ " ms\n");
	}

	public void report(long testTimeInNanos) {
		totalTime += testTimeInNanos;
		finishedTests++;

		if (testTimeInNanos > maxResponseTime) {
			maxResponseTime = testTimeInNanos;
```

```
        }
}
```

```
	private class Test implements Runnable {
	   Thread t;
	   int id;

	   public Test () {
		   t = new Thread(this, "Test");
		   t.start();
		   id = ++threadsCreated;
	   }

	   public void run() {
		   long start = System.nanoTime();

		   try {
			   client.send(URL, SOAP_REQUEST, null)
				   .getPayloadAsString();
		   } catch (Exception e) {
			   e.printStackTrace();
		   }

		   long end = System.nanoTime();

		   report(end - start);
	   }
	}
}
```