Access Point Framework – towards a more scalable ESB solution

SANJA JANKOLOVSKA

BOSHKO ZHERAJIKJ

Supervisor: Don Baldwin, CEO of Aurenav LLC.
Examiner: Mihhail Matskin

Degree project in Software Engineering of Distributed Systems

Stockholm, Sweden 2012
Abstract

The aim of this project was to investigate the potential issues of existing ESB products and present how the Access Point Framework (an Enterprise Application Integration Framework developed within Aurenav LLC.), mitigates these problems. For this purpose, the framework was compared with two existing ESB solutions- Mule ESB and BizTalk® ESB Toolkit. In order to be able to present the simplicity of integration with the Access Point Framework, first, it was moved to open source underlying technologies. The two most important ones were the Message Oriented Middleware products –ActiveMQ and RabbitMQ, both of which are interchangeable, depending on the user needs. Additionally, after having the framework fully functional, two simple applications were integrated to exemplify the ease of integration. Currently, the Access Point Framework is synchronous and single threaded, however, the necessary improvements are discussed throughout the paper. The steps described paved the way for carrying out the performance testing that was done on the framework. The project serves mainly as a comparison and evaluation document but also as a base point for moving the APF to an open source solution, while preserving the configurability in order to extend the functionality in later releases.
Sammanfattning

Acknowledgements

We would like to thank our supervisor, Don Baldwin, for giving us the opportunity to work on this project within Aurenav LLC., as well as his guidance, knowledge and experience sharing. We would also like to thank our examiner, Professor Mihhail Matskin, for his contribution, guidance and collaboration throughout the development of this project. We are thankful to our parents and families for giving us the opportunity to reach this point of life where we complete the Master degree with this project. And last but not least, to our friends that have been with us, giving their constant support. Thank you all!
Table of Contents

1. Introduction .......................................................................................................................... 1

2. Literature Review ................................................................................................................... 2
   2.1. Background ...................................................................................................................... 2
   2.2. Enterprise Application Integration (EAI) ......................................................................... 2
   2.3. Service Oriented Architecture (SOA) ............................................................................ 5
   2.4. Enterprise Service Bus (ESB) ......................................................................................... 7
       2.4.1. Basic Functionalities of an ESB .............................................................................. 7
       2.4.2. Characteristics of an ESB ....................................................................................... 8
   2.5. Normative Data ............................................................................................................... 10
       2.5.1. Canonical Model .................................................................................................... 10
       2.5.2. Enterprise Data Model .......................................................................................... 12
   2.6. Messaging Oriented Middleware (MOM) ....................................................................... 13
   2.7. Related work .................................................................................................................. 16
   2.8. Existing ESB Solutions ................................................................................................. 17

3. Problem Definition and Methodology .................................................................................... 19
   3.1. Problems of existing integration solutions ..................................................................... 19
   3.2. Goals and Objectives ..................................................................................................... 21
   3.3. Methodology .................................................................................................................. 22
       3.3.1. Research ................................................................................................................ 22
       3.3.2. Investigating existing ESBs and MOMs ................................................................. 23
       3.3.3. Choosing the ESB products .................................................................................... 23
       3.3.4. Comparison Criteria for the ESBs ......................................................................... 24
       3.3.5. Evaluating the APF ............................................................................................... 24
       3.3.6. Choosing the MOMs ............................................................................................. 25
       3.3.7. Feature evaluation for the MOMs ........................................................................... 26
       3.3.8. Evaluating the MOMs .......................................................................................... 26

4. The Access Point Framework .................................................................................................. 27
   4.1. Components of the APF .................................................................................................. 27
       4.1.1. Application Integration Component – AIC ............................................................ 27
4.1.2. Transformation and Validation Engine – TVE ........................................29
4.1.3. Messaging Platform Adapter Service – MPAS ...............................30
4.1.4. Repository Service – RS .................................................................32
4.1.5. Base Components .........................................................................33
4.1.6. Access Point – AP ..........................................................................33
4.1.7. Message Warehouse – MW ...........................................................34
4.2. APF Architecture ................................................................................35
4.3. Discussion ..........................................................................................39
5. Comparison with ESB Products ..............................................................41
  5.1. Mule Architecture ..............................................................................41
    5.1.1. Similarities with APF .................................................................44
    5.1.2. Differences with APF .................................................................48
  5.2. BizTalk® ESB Toolkit Architecture .....................................................52
    5.2.1. Similarities among architectures ...............................................57
    5.2.2. Differences with the APF ..........................................................59
6. Implementation .......................................................................................65
  6.1. Database Replacement .................................................................65
  6.2. Messaging Middleware Replacement ................................................66
    6.2.1. Feature Evaluation .................................................................66
      6.2.1.1. ActiveMQ .................................................................66
      6.2.1.2. RabbitMQ .................................................................67
    6.2.2. Implementation of Messaging Functionality ..................................69
      6.2.2.1. ActiveMQ .................................................................69
      6.2.2.2. RabbitMQ .................................................................70
  6.3. Application Implementation and Integration .......................................71
    6.3.1. Implementation .................................................................72
    6.3.2. Integration ..............................................................................73
7. Evaluation ...........................................................................................78
  7.1. Description of the benchmarking programs ......................................79
    7.1.1. Messaging solutions performance evaluation ..............................79
    7.1.2. APF performance evaluation ....................................................80
  7.2. The Test Process .............................................................................81
7.2.1. Message Queues ........................................................................................................... 82
7.2.2. Access Point Framework ............................................................................................ 82
7.3. Results .................................................................................................................................. 83
  7.3.1. Message Queues ........................................................................................................... 83
    7.3.1.1. Results from single machine .................................................................................. 83
    7.3.1.2. Results from two machines .................................................................................. 86
    7.3.1.3. Comparison one vs. two machines ....................................................................... 88
    7.3.1.4. Discussion on the Message Queues Performance ................................................. 90
  7.3.2. Access Point Framework Performance Results ............................................................ 90
    7.3.2.1. Discussion on the APF’s Performance ................................................................... 92
8. Conclusion ............................................................................................................................ 94
9. Future Work .......................................................................................................................... 95
Appendix A .............................................................................................................................. 101
Table of Figures

Figure 1: Point-to-Point Messaging Model (Queue) .......................................................... 13
Figure 2: Publish/Subscribe Messaging Model (Topic) .................................................. 14
Figure 3: Hub and Spoke Topology ................................................................................ 19
Figure 4: Bus Topology .................................................................................................. 20
Figure 5: MPAS Channels ............................................................................................... 32
Figure 6: Message Warehouse ....................................................................................... 34
Figure 7: APF Architecture ............................................................................................ 35
Figure 8: Centralized Integration .................................................................................... 37
Figure 9: Distributed Integration; EAI Approach ............................................................ 37
Figure 10: Fully Distributed Integration ......................................................................... 38
Figure 11: Deployment Models functioning together ..................................................... 38
Figure 12: Mule ESB Architecture .................................................................................. 41
Figure 13: Message Flow within Mule ESB .................................................................. 43
Figure 14: Enterprise Service Network Topology with Mule ......................................... 46
Figure 15: APF ESB topology ......................................................................................... 47
Figure 16: APF Pipeline Topology .................................................................................. 47
Figure 17: BizTalk® Server Architecture ....................................................................... 53
Figure 18: BizTalk® ESB Toolkit ................................................................................... 55
Figure 19: BizTalk® and ESB Toolkit components ......................................................... 56
Figure 20: BizTalk® Server Hosts and Host Instances .................................................... 59
Figure 21: BizTalk® SQL Server Topologies .................................................................. 62
Figure 22: JMS Programming model Overview ............................................................. 69
Figure 23: Message Flow ............................................................................................... 77
Figure 24: Throughput measured for ActiveMQ and RabbitMQ ................................. 83
Figure 25: Average Lattency measured for ActiveMQ and RabbitMQ ......................... 84
Figure 26: Execution Time measured for ActiveMQ and RabbitMQ ............................ 84
Figure 27: Non-Persistent mode for ActiveMQ and RabbitMQ ..................................... 85
Figure 28: Persistent mode for ActiveMQ and RabbitMQ ............................................. 86
Figure 29: Throughput for consumer and producer on separate machines ................... 86
Figure 30: Execution time for producer and consumer on separate machines ............. 87
Figure 31: Execution flow for non-persistent consumer ................................................ 87
Figure 32: Execution flow for persistent consumer ....................................................... 88
Figure 33: Persistent consumer on ActiveMQ-comparison on local and two machines ... 88
Figure 34: Persistent consumer on RabbitMQ-comparison on local and two machines ... 89
Figure 35: Non-persistent consumers-comparison on local and two machines .......... 89
Figure 36: Performance results for the Requester and Provider ..................................... 92
1. Introduction

Nowadays, there is a great number of ESB products on the market, which facilitate the process of application integration within an enterprise. All of them offer the core ESB functionalities, and differentiate themselves mainly by their architecture. However, there are some common issues which arise when the throughput that needs to be handled by the ESB product drastically increases. Due to the complexity of some of the architectures, scalability is not their strongest point. Adding extra ESB instances does not only increase the cost, but also the maintenance becomes difficult to handle.

This is a common problem that arises in organizations with enormous number of transactions, and there are also similar problems that mainly arise from the architecture of the ESB products and their tight coupling between the building components. Of course, there is no perfect product when it comes to these issues, however, the Access Point Framework (APF), tries to address and mitigate them as much as possible through its flexible and simple design with loosely coupled components, easily pluggable and interchangeable.

The APF is an enterprise application integration framework, which offers simple and elegant design, being built on the foundations of scalability, as the traffic within the enterprise increases. Therefore, the motivation behind this project would be to show how the APF offers solution to the existing problems, in comparison with specific ESB products, currently available on the market.

Besides this main purpose, throughout the project, the APF will be migrated to completely open source underlying technologies, in order to fully offer the benefit of low cost. As part of this process, two underlying messaging solutions will be integrated into the framework, and a comparison will be provided for the future users, to facilitate the choice of the more suitable messaging technology that will fit their needs.

The work done will be presented in this paper in the following order: Chapter 2 will provide all the background information, necessary for understanding the topic and the details discussed through the rest of the paper. Following will be Chapter 3 where the problem definition will be presented, together with the main goals and objectives accompanied with the methodology approach for achieving them. In Chapter 4 there will be a detailed presentation of the APF’s architecture and its building components. Chapter 5 will concentrate on the comparisons between the APF and the two chosen ESB products from the market. Further, Chapter 6 will discuss the implementation work done throughout the project, while Chapter 7 will contain the evaluation of the performance of both the APF and the integrated messaging solutions. The discussion will be finalized with some ideas about the potential future work and conclusion of the paper.
2. Literature Review

2.1. Background

It is rarely the case that a single application fits the needs for an enterprise. Thus, typically several best of breed proprietary software applications are used by an enterprise for successful operation. When using third party software, it might sometimes be impossible to change the data format of what the application is using. Legacy applications might prove still useful for enterprises and also it would be more expensive to develop everything from scratch. Thus, many businesses might add new functionality or connect various applications, depending on the need, in an ad-hoc manner, leading to an accidental architecture rather than having a standardized way of integrating on an enterprise level (Chappell, 2004; Hohpe & Woolf, 2004). At the same time, the world of business is changing rapidly and many applications require change in how business processes are performed. It is required to be able to add new or merge existing applications for achieving better performance but also for the business to stay competitive. In other cases the solution might be as simple as acquiring another application that might solve the problem of integrating, but not always that is the case.

2.2. Enterprise Application Integration (EAI)

Enterprise application integration aims at merging and joining in various standalone applications or islands of functionality so that it can bring more value to the business. According to Linthicum, application integration is a strategic approach to binding many information systems together, at both the service and information levels, supporting their ability to exchange information and leverage processes in real time (Linthicum, 2000). In addition, according to (Lee, 2003), one can say that “EAI is a business computing term for plans, methods, and tools aimed at modernizing, consolidating, and coordinating the overall computer functionality in an enterprise”. Key objectives include connecting existing applications and making it possible to exchange data and processes across departments; this includes message acceptance, transformation, translation, routing, message delivery and business process management (Goel, 2006) which will be described later on. Besides these, it would also be desirable to offer better performance and scalability capabilities (Baldwin, 2012).

Integration should be achieved with as less programming and change to existing legacy applications as possible (Hohpe & Woolf, 2004). For this matter, another key term when it comes to EAI is loose coupling. The traditional approach to establishing communication between applications would be through the use of the RPC manner of programming that requires for each application to know what are the methods and the parameters that each method accepts. This is highly undesirable when it comes to EAI, since there is a need for
applications to communicate without the need of knowing all the details required by other applications (Chappell, 2004; Hohpe & Woolf, 2004). Also as soon as one application is required to be changed it would be hard for other applications to communicate since a method or the parameters might change as well. This would make the process of integration of applications harder as more change would be required.

Having established that loose coupling is highly desirable and needed when it comes to EAI, further on, the necessary elements for achieving loose coupling can be distinguished. When it comes to providing a loosely coupled solution two elements are essential. First is the messaging channel that might utilize different technologies (for instance TCP/IP) that will be used to establish the communication between different applications (Hohpe & Woolf, 2004). The channel will be used to establish the necessary communication between various applications. Any data used by the collaborating applications can use the channel to communicate agreed set of messages that would contain the necessary data used by the concerned applications (Hohpe & Woolf, 2004). These two elements establish the basis for integrating two applications.

Typically there would be several disparate applications that need to communicate. Each of these applications might have a different representation format of the data that it’s using, making it harder for applications to communicate. Thus, what is needed is for the middleware to actually translate from one application format to another so that it would be understandable by the collaborating applications (Hohpe & Woolf, 2004).

When it comes to integrating more than two applications, one other aspect that needs to be taken in consideration is routing (Hohpe & Woolf, 2004). This is hard to achieve as more and more systems are integrated, if the initial approach was incorporating the information about other systems in one system. For instance, if there is one system responsible for keeping employee information and whenever new information was added the system would be responsible for disseminating the information to the integrated systems. If there was a new application added, the system would need to be updated. Thus, it would be desirable for the middleware to handle the routing of the messages as well.

Besides the named elements, another important aspect is the systems management function (Hohpe & Woolf, 2004). The systems management function would ensure that all systems are running but also would be responsible for error handling and reports generation regarding the communication between various applications and components of the integration solution. Last but not least, having in mind that legacy, custom and packaged applications are not done in mind for achieving integration, what would be required is a message endpoint, to be able to connect the necessary application to the communication channel (Hohpe & Woolf, 2004).

Having identified the necessary elements that are required and typically encompass EAI, three integration styles can now be identified. Woolf and Hohpe (Hohpe & Woolf, 2004) suggest 4, these are file transfer, shared database, method and messaging level integration, while (Linthicum, 2000) states, at the data level, method, application interface
and Graphical User Interface (GUI) level. Based on the observations, three clear levels can be distinguished, which are integration at the data level, messaging level and GUI level. At the data level or shared database, integration is achieved through the use of technologies used for transforming and moving between data stores or databases (ex. ETL tools). GUI level encompasses the use of technology and techniques at using the information from the GUI as a common integration point (ex. screen scraping).

Last but not least, the messaging approach is what is going to be considered in a more detailed manner throughout this paper, based on both (Hohpe & Woolf, 2004; Baldwin, 2012) is considered as the most suitable approach to EAI. Asynchronous messaging is the main enabler for achieving loosely coupled interfaces (Chappell, 2004). Through the use of messages, only the concerned applications or services will be able to inspect and process the data that is of interest of the applications. Applications can remain as independent as possible by just exposing their interfaces and at the same time, when a calling application sends a message it does not need to wait instantly for a reply. It can continue performing its function knowing that when the called application (service) will be finished it will be notified for further processing. This might not always be desired or preferred but it does alleviate some problems like the necessity of having two systems up and ready at the same time. Asynchronous communication does have a downside, which is the complexity that surrounds it when designing, developing and debugging (Chappell, 2004; Hohpe & Woolf, 2004). Messaging also enables transformation during transit that is transparent to the applications that are using the data. This gives greater flexibility when it comes to making decision regarding how messages are being transferred among applications. However, these transformations could possibly create overhead or require for a lot of “glue” code for fitting everything together (Hohpe & Woolf, 2004).

Through the description one can see that EAI as a concept is very promising, however the actual reality is rather different, since typically the application integration is done through the use of hub and spoke integration brokers or EAI hubs (Chappell, 2004), implemented as Message Brokers. Having the hub and spoke architecture, integration is achieved by having a centralized mechanism for achieving integration between applications which is better than just merging everything in an ad-hoc manner. Another positive aspect is having an improved separation between business processes and integration code (Chappell, 2004). In an EAI hub, the main elements are included by having application adapters, data transformations, routing and connectivity functionalities. However, since it is a centralized approach every message needs to travel to a central hub that actually performs the processing and instructs the following routing destinations (Chappell, 2004). Another architectural pattern typically employed is the bus, more specifically through the use of an Enterprise Service Bus (ESB) products. The ESB pattern will be discussed in further detail in section 2.4.

In order to resolve the problem of tightly coupled applications, EAI aims at abstracting away the tight coupling by allowing applications to communicate in an asynchronous manner through messages introducing loosely coupled interfaces that are easier to integrate. The goal is to make the systems as loosely coupled as possible, but at the same time reasonably responsive and capable to share data (Hohpe & Woolf, 2004). One approach that has grabbed
the attention is the Service Oriented Architecture that will be presented in the following sections.

2.3. **Service Oriented Architecture (SOA)**

Having EAI as an established business need of integrating applications, this section is going to elaborate the concept of Service Oriented Architecture (SOA). By the end of this section the aim is to have a clear distinction of how the two are related but also where the ESB as a product relates to the two. The Service Oriented Computing paradigm together with Web Services technologies have brought great interest in SOA (High, 2005; Keen, 2004).

Before proceeding with the explanations and definitions of the SOA, there needs to be a clear distinction on what constitutes a service. According to Keen, M., et al.: the most commonly agreed-on aspects of the definition of a service in SOA are (Keen, 2004):

- "Services are defined by explicit, implementation-independent interfaces.
- Services are loosely bound and invoked through communication protocols that stress location transparency and interoperability.
- Services encapsulate reusable business function."

One other aspect that can be added is that in order for a function to become a service it should provide some value for the business or the business process (Lankhorst, 2005). Unless the service brings some value one can run into the problem of defining everything as a service. When it comes to services, typically the internal processes are kept as a black box and the service is only accessed through the interface since the consumer is only interested in the quality and the functionality that is going to be delivered by the service.

By specifying services as individual, self-containing pieces of functionality, the aim is to offer greater flexibility and loose coupling between components. The desirability can be related to the EAI concepts described in the previous section. Having the business functionalities defined as disparate components offers yet another benefit of grouping the functionalities as needed, rather than them becoming tightly related to the specific implementation. This leads to the term service orientation where the set of business processes that are linked through the use of services, compose an integration approach within the business (High, 2005).

The concept that encapsulates this behaviour is Service Oriented Architecture (SOA), that is

"In an SOA, all software components (or functional units that are visible for other entities to invoke or consume over the network) are modeled as services. That is, the architectural premise is that all business tasks or business processes that are built in software are designed as services to be consumed over a network (Goel, 2006)."
SOA can be an architectural style for creating an Enterprise IT Architecture that exploits the principles of service orientation to achieve a tighter relationship between the business and the information systems that support the business (High, 2005). In this manner one can achieve a composite application that represents the integrated services that support a particular business process. It can be interpreted also as a set of design principles that enable units of functionality to be provided and consumed as services (Lankhorst, 2005). The key to SOA is that the service concept can be equally applied to both applications but also business.

In addition, Web services as a technology is a key enabler and has been widely adopted for realizing the concepts of SOA, even though it is not the only way of achieving it. It is important to note that Web services can be used both in an intra-network and over the Internet. A Web service is a software system that supports machine to machine interaction through the use of a set of technologies including the use of XML, HTTP, SOAP, UDDI and WSDL but also other Web related standards (Graham, 2004).

There are additionally three roles in a SOA, which are the service provider, service requester and service broker (Graham, 2004). The service provider is responsible for making access to a service description so that other services can access it, publishing the description in several service registries and accepting invocation messages from several service requestors. Service requester takes the responsibility of discovering a service description that is published in a service registry but also is responsible for the invocation of the Web service hosted by the service provider. Service registry is responsible for the advertisement of various service descriptions that are published by the service providers and allowing the service requesters to search for service descriptions.

Moreover, there are three operations that are included in SOA that define the interactions between the three roles defined previously, namely, publish, find and bind (Graham, 2004). The publish operation is when the service provider contracts with the service registry to advertise a service description that can later on be called by a service requestor. The find operation is a contract between the service requestor and the service registry for discovering a service that has been registered, with certain criteria (ex. type of service, quality). The registry then provides the requestor with a list of found service descriptions that match the criteria of the requestor. Last but not least, the bind operation marks the contract between the service requestor and service provider, enabling the service requestor to use the provided services.

Even though there are similarities between SOA and EAI, the main difference is how the separation is achieved in order to provide loose coupling. One can see that through the use of SOA, one could possibly achieve integration. SOA concentrates on the design and development of software as a service, whereas EAI is a more generic approach to application integration. Web services and SOA could be one way of achieving application integration, but traditional Message Oriented Middleware, message brokers and others are also tightly related to EAI. While SOA aims at describing how software is going to be build, EAI concentrates on building an architecture within and across the enterprise that will allow for
flexible inclusion of various applications and services, not just Web services, but also their coordination.

### 2.4. Enterprise Service Bus (ESB)

When it comes to SOA there are many ways of realizing it, however the Enterprise Service Bus architectural pattern can serve as one example that can achieve a SOA. It uses some of the concepts that were discussed in EAI section in combination with Web services, SOA and other standards, to provide a common infrastructure backbone for services in a SOA (Chappell, 2004; High 2005). The bus actually serves as a communication medium between services, but also between clients and services rather than just establishing point-to-point communication (Desmet, 2007). The bus is actually similar to a communication bus in a computer system, and serves as the main point in enabling communication between CPU, memory, and peripherals (Hohpe & Woolf, 2004). The difference is that the ESB is a software abstraction or version of the system bus. Moreover, the ESB pattern takes on some of the practices described in the EAI section together with SOA standards to establish the basis for a more flexible architecture for accommodating future business changes and the integration of loosely coupled applications.

An ESB can be understood as a layer within the enterprise architecture, which is added on top of the messaging system (DiMaggio, 2008). A more formal definition of an ESB is that it is a “standards based integration platform that combines messaging, web services, data transformation and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity”. The definition will be clearer within the following sections where the core functionalities of an ESB will be discussed together with some characteristics of an ESB.

#### 2.4.1. Basic Functionalities of an ESB

There are certain functionalities that are essential for an ESB and are expected to be provided by it. They facilitate the integration process and contribute for the inter-operation of the connected applications. Considering the fact that an ESB is a specialized version of EAI, it can be said that these functionalities map to the ones discussed in section 2.2. Of course, different ESB realizations handle their implementation in various ways, but still maintain them as core functionality.

In that manner, some applications, which are not developed initially to communicate with other applications or with the existing messaging system used within the ESB, are not so straightforward to integrate. For that purpose adapters are introduced, which are used as interface to the messaging bus, and correspond to the concept of “endpoint” discussed in the Enterprise Application Integration section. The only difference between the two is that the endpoint is integrated within the application, while the adapter is a separate component
outside the application code. The adapters are used to convert the incoming messages to the application into its proprietary format and invoke the necessary methods through the application’s API. In addition, the adapters are listening to the internal events of the applications and putting the corresponding outgoing messages on the communication channel (Hohpe & Woolf, 2004).

In addition, one must take into consideration the fact that the applications which will be plugged in the ESB will not use the same communication protocols. Therefore, the ESB is required to provide the support for the variety of protocols and facilitate the communication between services with different protocols such as HTTP, SOAP, REST, FTP, etc. (Ibm.com, 2007; Hohpe & Woolf, 2004)

Furthermore, due to the versatility of the applications, it is common that they use different formats of representing the information they work with. In order to surpass the differences, data transformations are provided as functionality, which were briefly introduced in section 2.2 and are further elaborated in subsection 2.5.1 on Canonical Model.

In the lead, another important aspect of the ESB is its routing functionality mentioned in section 2.2. Routing reduces the necessity of the service requesters knowing about the location of service providers, by handling it separately. One advantage of the ESB compared to the centralized EAI broker, when it comes to routing, is that within the ESB there is no centralized engine responsible to relaying all the messages. Instead, specialized services take care of the message routing. One type of message routing common for an ESB is the so called Content-Based Routing. It is characterized by the fact that it uses the message data in order to find out where to route the message next. In addition, there is also the Itinerary-Based Routing where the messages carry as metadata inside their headers, the intermediary addresses where they have to pass from, in order to reach their final destination (Chappell, 2004).

Besides the adapters, transformations and routing, there are additional functionalities which are characteristic for ESBs such as orchestration of services, support for an UDDI registry for locating and registering the services, support of essential security features such as authentication, authorization of applications and several more which are briefly elaborated in the following section as characteristics of an ESB (Kanetkar, 2006).

2.4.2. Characteristics of an ESB

The following section presents a summary of the fundamentals of an ESB, from the overlapping points of view of several professionals (Menge, 2007; Woolley, 2006, Chappell, 2004). There are certain characteristics that describe the functionalities which an ESB should provide, as well as how should it be designed and what advantages should it provide to the enterprise, if developed correctly. These characteristics are implemented with different levels of quality in the various existing ESB products, and will facilitate later on in the evaluation of these ESBs. Below is provided short elaboration of each feature:
• **Invocation, routing and mediation** - these are the three essential functionalities of an ESB, meaning that it will send requests and receive responses from service requesters to service providers, route the messages conveniently and transform them to different formats understandable by the services (Menge, 2007; Woolley, 2006);

• **“Basic Connectivity”** - should provide the possibility for new services using different technologies to be connected to the bus through the use of connectors such as J2EE Connectors, JMS, etc. (Cragsgs, 2003; Chappell, 2004);

• **“Pervasiveness”** - can be used to integrate applications all over the enterprise, on a global level, despite the geographical limitations and different application implementations (Chappell, 2004);

• **“Standards-Based Integration”** - ESBs should use and follow certain industry standards such as XML, for instance, to enable message transformation, or SOAP, WSDL, UDDI for web service communication, in order to simplify the integration process (Menge, 2007; Chappell, 2004);

• **“Highly Distributed Integration and Selective Deployment”** - ESBs should provide flexibility and scalability of message routing, transformation and similar integration services, by allowing them, at the same time, to operate independently and in transparent, distributed manner (Cragsgs, 2003; Chappell, 2004);

• **“Distributed Data Transformation”** - ESB should allow transformation from one data format to another, and the services responsible for the data transformation could be located and accessible from anywhere on the bus (Chappell, 2004);

• **“Extensibility through layered services”** - specialized processes can be separated in layered services, where each layer will have certain responsibilities which can be extended by adding more layers (Chappell, 2004);

• **“Event-driven SOA”** - the ESB exposes the applications and services to asynchronous events and hides the protocol and routing details from them; therefore, whenever they receive event as a message, they need to process it, without having to take care of the low level details of communicating with the other services (Menge, 2007; Chappell, 2004);

• **“Process Flow”** - The ESB is capable of orchestrating the process flow of simple and complex business processes using BPEL4WS or the messages’ metadata (Menge, 2007; Chappell, 2004);

• **“Security and Reliability”** - There should be a significant level of security support by the ESB, between itself and the applications, as well as between the ESB nodes themselves; this includes authentication, authorization, cryptography, etc. In addition, reliability is achieved by having Message Oriented Middleware within the ESB which will guarantee reliable data delivery (Menge, 2007; Chappell, 2004);

• **“Autonomous but Federated Environment”** - there is the necessity for different departments to take care of their data and business processes independently, without having to pass all the traffic through a centralized message broker. However, certain information need to be shared throughout the enterprise. For this purpose the ESB has to provide the ability for separate business units to operate autonomously, but still be able to share data on an enterprise level and compose a federated unity with the rest of the departments (Chappell, 2004);
“Remote Configuration and Management”- it is not always necessary and possible to have an IT team on each location to manage the applications for that specific unit. For that purpose, it is necessary for the ESB to provide off-site management and configuration. This can be usually supplemented with logging, auditing and controlling process solutions (Menge, 2007; Chappell, 2004);

“XML as the Native Data type of the ESB”- as discussed by Chappel, XML is the “ideal” format for storing data that are to be exchanged between the integrated applications (Woolley, 2006). Due to the many benefits of using XML, such as contributing to the self-descriptiveness of the data structures, it should be considered as the native data type for an ESB (Craggs, 2003; Chappell, 2004);

“Real-Time Throughput of Business Data”- the ESB should provide real-time throughput of the data exchanged between the applications in order to avoid integration methods such as batch processing which are associated with certain problems (being error and delay prone) (Chappell, 2004);

“Operational Awareness”- the ESB should facilitate a business analyst to check the operational state of the business processes ongoing within the ESB. This can be facilitated by adding “tracking points” which collect data that passes through and can be later revisited (Chappell, 2004);

“Incremental Adoption”- the ESB should have the capability of being incrementally extended with new applications and services (Chappell, 2004);

2.5. Normative Data

When it comes to application integration, it is highly desirable that all business information within an enterprise to be represented in a standardized manner across the enterprise in order to facilitate the integration process. This would eliminate any duplication and reduce redundancy while integrating applications or building services, for instance. Thus, normative data conforms to the standardized way of representing the information within the enterprise (Baldwin, 2012). A way of achieving normative data within the enterprise is through a Canonical model, or in case there are several Canonical models, through an Enterprise Data Model. The following sections provide a more elaborate description of these two concepts.

2.5.1. Canonical Model

Considering the variety of applications that are plugged into the ESB, a common consequence is the fact that these applications have different data representations, which creates a difficulty when they have to “talk” to each other (Chappell, 2004). An example of such scenario might be the way two applications represent information about a person. One application might represent the name and surname in one field, while the other has two separate fields for that purpose. Therefore, a problem arises when one of the applications needs to communicate the person’s data to the other, having in mind that they are not representing the information in the same way (Ibm.com, 2008a).
In order to enable the applications to “understand” each other’s representational formats, there exist message translators, which translate one format to another, understandable by the application. One way of implementing this solution is having a message translator for each communication channel from one application to another. However, the number of translators will rise with the number of applications according to the formula $n(n - 1)$. This means that if 5 applications are plugged in the bus, 20 translators will be needed, which is quite a high number and will become even more significant when the number of applications increases (Hohpe & Woolf, 2004).

In that manner, a more suitable approach, especially for enterprises with higher number of applications, is creating a Canonical Data Model which represents the information’s structure within the enterprise (Ibm.com, 2008b). Therefore, in order to avoid having translators for every channel between applications, the applications sending out the data will have to translate it into the Canonical Data Format and then the services which are supposed to receive the message will transform it from the canonical to the application specific format in order to further process it. In order to facilitate the creation of the Canonical Data Model within the organization, certain standard specifications can be used as foundations of the model such as HL7 in health care or FIX in financial services (Chappell, 2004).

However, one downside of using the Canonical Data Model is the fact that it requires double transformation, meaning that the sending application needs to transform the data to the canonical format and then the receiving application needs to transform it again from the canonical to its specific data format. Sometimes, when there are not a high number of applications, developing a canonical data model can be impractical, but as the number of application increases, the number of formats increases as well. This elevates the complexity of the integration process, since for every new application a translator to every other application is required. Nevertheless, if the canonical data model exists, the new application needs only a single translator to the canonical data model (Hohpe & Woolf, 2004).

In the lead, the canonical data model is further used to develop the canonical message model (CMM). The CMM represents the “standardized format” for exchanging the data on the ESB. It usually consists of types, attributes and elements representing the business objects (Ibm.com, 2008b). According to (Ibm.com, 2008a), there are four approaches of implementing a CMM in an ESB. One is the already discussed approach, where each requester service translates its specific message model to the CMM and then the provider translates from the CMM to its proprietary message model. An important characteristic of this approach is that the translation occurs within the ESB itself (Ibm.com, 2008a).

The second approach is having each provider/requester translate to and from a CMM compatible message within its own environment. The difference of this approach is that the transformation is not happening centrally within the ESB, which in this scenario is only responsible for routing the messages. This is a potential benefit of the approach, having the load of transformations removed from the ESB, however a negative aspect is that the transformations need to be implemented at the application side and in case of not-well documented legacy applications, this becomes too difficult to handle (Ibm.com, 2008a).
The third, and, as referred to, “ideal approach” advocates that all services have been defined using the CMM, therefore, there is no necessity of transformations in the ESB, nor on the applications’ side. The benefits of this approach are quite visible, when it comes to developing new applications, which are to be plugged into the ESB, since they can be developed to incorporate the CMM and completely removing the load of transformations. On the other hand, there are also some negative sides of this scenario. Having in mind that not all applications that need to be plugged already conform to the CMM, changes within their implementation will have to be done, and this becomes too expensive, especially with legacy applications (Hohpe & Woolf, 2004). Therefore, the suitable situations for implementing this approach are quite rare (Ibm.com, 2008a).

Last but not least, is the hybrid approach which combines all of the previously discussed scenarios. Some of the services can be better off without a CMM model, having a translator on their connection which will translate the messages between those two. Other services connected on the bus which will be difficult to modify will require the transformation to take place within the ESB. Others may be easier to modify, so will remove the transformation load from the ESB and move it in the service environment. And finally, some might have already implemented the CMM, so they will require no transformation. This way, the hybrid approach combines all the advantages from the previous solutions and produces the best result that will serve the organization most optimally. The only negative aspect of it is that for every new application that is to be plugged, a decision has to be made of which transformation approach will it use. This can be quite time consuming in environments when a lot of new applications are being introduced to the ESB constantly (Ibm.com, 2008a).

2.5.2. Enterprise Data Model

There are certain situations when there are several CMMs established already within the enterprise and there is the necessity of the applications to communicate with each other, although they belong to different domains (and thus different CMMs). For that purpose, the Enterprise Data Model comes in place, which can consist of several CMMs. It is a model which creates an “integrated view” of the data within the enterprise and enables for more than one canonical data model to coexist (Tdan.com, 2005). An example of how enterprise data model comes into practice can be observed, if an organization where group of applications conform to HL7 and another group conforms to FIX, need to communicate to each other. In such case, the Enterprise Data Model standardizes the data exchange between these two different domains.
2.6. Messaging Oriented Middleware (MOM)

Message Oriented Middleware (MOM) is the underlying mechanism used by the ESB for relaying messages between the participating applications in the enterprise architecture. MOM separates the location details of the message senders and receivers, and therefore, they don’t need to be aware of each other. The communication between these two entities happens through the messaging system, which is usually a message broker that takes care of the message routing. Thus, whenever a message producer needs to send a message to message consumer, it uses the Messaging API to pass the message to the message client which further sends it to the message system. The message system identifies, from the message header, where to route the message, finally sending it to the message client of the designated message consumer. This way the communication takes place through the MOM (Curry, 2004; Chappell, 2004).

The two most common messaging models are point-to-point and publish-and-subscribe. In point-to-point, a producer sends a message to a queue, destined to only one consumer. Several consumers could exist, but the message is consumed only by one (Curry, 2004). Figure 1 depicts the model:

![Figure 1: Point-to-Point Messaging Model (Queue)](image1)

In a publish-and-subscribe model (Figure 2), several consumers are subscribed on a certain topic and each receives a copy of the message published by the producer of that topic (Chappell, 2004).

---

1 Image taken from: (Curry, 2004)
The two messaging models are the key abstractions that enable decoupling and asynchrony among communicating applications.

In the lead, loosely coupled interfaces and asynchronous interactions are the most important aspects of the bus (Chappell, 2004). Thus, MOM provides the opportunity for loosely coupled environment, where the applications do not need to know how to access each other's interfaces, since all this is handled by the middleware. In addition, loose coupling contributes to detaching the performance of the separate subsystems from the overall performance throughout the enterprise since the scalability for example of one system, does not need to affect the others (Curry, 2004).

Accordingly, asynchronous interactions facilitate the communication operations to be “standalone units of work” between the applications (Chappell, 2004). They enable the producer to just create a request, and continue working, expecting in the meanwhile to get a response without having to block and wait for it. In asynchronous messaging, there is still the need for reliability, which is typically employed by the MOM itself (Chappell, 2004). In order to ensure reliability, two concepts are important, which are persistence mode and acknowledgements.

In that manner, the messages that are passing through the MOM are marked either as persistent or nonpersistent. Having a message being marked as persistent, means that the message will stay in the queue and that the MOM needs to save the message to a disk. Having the persistent mode enabled, if the intended consumer or message system fails, later on when the consumer or system is running again, the message would be redelivered or forwarded. On the other hand, non persistent messages are kept only in the queue/topic. In case of failure or recovery of the messaging server or consumer, there is no guarantee that the message would be delivered to the intended recipients, thus the message could be potentially lost (Chappell, 2004).

When it comes to reliability, another important concept is message acknowledgements (Chappell, 2004). This is how the messaging server is aware whether a message has been delivered successfully to a consumer or not. The producer of messages typically sends of a message to the server, having the MOM guarantee delivery rather than blocking and waiting

---

2 Image taken from: (Curry, 2004)
for a reply. The server then awaits an acknowledgement from the consumer to ensure that the message has been consumed and notifies the producer of the message.

Overall, it can be said that through loose coupling and asynchronous delivery, MOM offers several advantages. One is avoiding the rapidly increasing number of interfaces, common for tightly coupled architecture, where usually the applications need to have point to point interface in order to communicate with each other, and the number of interfaces is usually $\frac{n(n-1)}{2}$, where $n$ is the number of nodes. In the case of loose coupling this exponential growth is replaced with linear, since only one interface is necessary for each application to be able to communicate with the middleware. Accordingly, the asynchronous messaging reduces the time necessary for the system to perform its operations since the producers do not need to block and wait for response, but they can continue executing while expecting it (Chappell, 2004).

There are number of both commercial and open source solutions when it comes to message oriented middleware. Some of the most well known are WebSphere MQ, TIBCO, SonicMQ and many other which are all proprietary systems that cause vendor lock-in (Curry, 2006). In addition, there is a great amount of problems when it comes to interoperability among MOM solutions, due to lack of standardization. A number of standards have emerged though, having Java Messaging Service as one widely established in the market. The notable difference is that JMS is an API, which by itself is an interface which does not provide any standardization. Its implementations vary among providers of the MOM solution, thus having no standardized protocol blocks interoperability among messaging systems (Chappell, 2004). This arguably will again cause vendor lock-in as well not elevating the problem of message systems interoperability. There might be afterwards a requirement for building a bridge between the two MOM systems (Chappell, 2004).

There are number of other protocols that try to solve the messaging problem such as, XMPP\(^3\) which is more appropriate for chatting systems, and also the two more important ones namely, STOMP\(^4\) and AMQP\(^5\) that strive for solving messaging interoperability. STOMP is also a wire level protocol, which has certain popularity due to its simplicity, which could impose as a change in the MOM world in the future. However, another protocol that is gaining market share, having a clearer goal of achieving high performance, flexibility, routing and delivery when it comes to messaging, is the Advanced Message Queuing Protocol (AMQP) (Vinoski, 2006).

In 2006, a working group was created between JPMorgan Chace (JPMC), Cisco systems, Envoy Technologies, Red Hat, Twist Process innovations and 29West which officially announced the formation of the Advanced Message Queuing Protocol (AMQP) (Vinoski, 2006). The goal of the working group is standardizing the wire format for allowing communication between different MOM systems. The details regarding these protocols are

\(^3\) [http://xmpp.org/]
\(^4\) [http://stomp.github.com/]
\(^5\) [http://www.amqp.org/]
out of the scope of this work. However, they are considered as an important factor when it comes to message oriented middleware. If a standardized protocol for interoperability among MOM exists, it would too contribute to enterprise application integration and ESBs since the MOM acts as the backbone technology.

2.7. Related work

There is a substantial research that can be utilized when it comes to evaluating existing ESB products, according to specific criteria, and comparing them. One of the leading studies is done by Forrester Research, with their latest release on Enterprise Service Bus products evaluation for the second quarter of 2011 (Vollmer, 2011). They have performed the evaluation based on 109 criteria for number of commercial and open source products, summarizing it by defining the leading ones on the market. The evaluation criteria had been divided in three major groups: “current offering” (including orchestration, mediation, change and control, architecture, connection), “strategy” (in terms of vendor’s cost, product strategy, customer references, strategic alliances) and “market presence” (in terms of ESB revenue, new customers, etc.) (Vollmer, 2011). In that manner, overall can be said that the evaluation of Forrester was done taking into consideration numerous factors that would be useful for companies to also decide which vendor to go with, while this thesis research will be concentrated more on the technical issues of the ESB products implementations.

Furthermore, equally significant evaluation and comparison work has been done by (Woolley, 2006), although quite specialized for the State of Utah. Nevertheless, he offers interesting criteria for evaluation such as compliance with industry standards, ease of integration, security feature offering and several more.

In the lead, an interesting and closely related work is done by Desmet et.al, in their paper “Throughput of different ESB approaches”, where they have evaluated two commercial and two open source products, taking into consideration their architecture and implementation, provided ready-to-use features, and ease of use (Desmet, 2007). In addition, they have experimented with performance testing of the ESB products, which could complement this project in terms of providing ideas for how to convey the performance testing for the Access Point Framework. In that manner, the paper “Early Capacity Testing of an Enterprise Service Bus” can be also useful. It discusses the necessity of capacity testing in the early stages of the projects, but it also presents how it is done. It outlines the requirements and the testing conditions, such as putting the ESB under high load and introducing an enormous amount of messages passing through, in order to evaluate the performance (Ueno & Tatsubori, 2006).

Of similar interest is the paper “Evaluating Open Source Enterprise Service Buses”, where the authors have evaluated three open source implementations, according to the features they support and their performance, in terms of response time for invoking external services. They have conveyed two experiments with different conditions for testing the
performance. Accordingly, this can be as well quite relevant background for the proposed evaluation to be done further on within this project (García-Jiménez, 2010).

Another elaborate research was presented by (Macvittie, 2006) where 8 ESB products were rated and compared in order to find the one which in total provides the best service as an ESB. The evaluation was done based on the “core bus features”, such as routing, transformation, orchestration, protocol support and management; also, into consideration were taken “integration” feature support as adapter support, ease of web service integration and their management; and last but not least was the price of the products.

So overall, having the background of how ESB products have been evaluated, compared and having the results from the comparisons, helped in establishing the basis for the future research. These papers will be guiding points in the future analysis of the ESB products, as well as the performance testing of the APF.

2.8. Existing ESB Solutions

There are number of both open source and proprietary products in the market that are presented as Enterprise Service Bus products. Each of these offer a various set of features and have different characteristics that encapsulate a different implementation to the concept of an ESB. When it comes to proprietary ESB products one can say that they originate from traditional EAI technologies and other proprietary software used for transformation and message routing capabilities (Rademakers & Dirksen, 2008). This leads typically to vendor locked solutions making it even harder to achieve integration but also can pose as a problem since many root in EAI principles of having message brokers or centralized functionalities (Baldwin, 2012; Rademakers & Dirksen, 2008). According to Forrester, commercial ESB products are being offered by IBM, Oracle, Progress Software, Software AG, Tibco (Vollmer, 2011). Their respective products are Websphere Enterprise Service Bus (WESB), Websphere Enterprise Service Bus Registry Edition (WESBRE), WebSphere Message Broker (WMB) provided by IBM, Oracle Service Bus by Oracle, Sonic ESB by Progress Software, webMethods ESB Platform by Software AG and ActiveMatrix Service Bus by Tibco, also Microfort BizTalk® ESB Toolkit and others.

Besides the existing commercial products there are also a number of open source ESB products that could be a solution due to the pricing of proprietary software as discussed further in the problem definition (Baldwin, 2012; Vollmer, 2011). In addition, according to Forrester, open source ESBs are said to offer greater functionality than previous years and are catching up with existing commercial solutions. The advantage also of open source solutions is their use of open standards such as Java Message Service (JMS), XML, J2EE Connector Architecture (JCA), and web services standards (Rademakers & Dirksen, 2008). Many open source solutions are based upon open integration specifications such as JBI or JMS. Some of the most notable open source ESB projects such as FuseSource6, MuleSoft7, Red Hat®

6 http://fusesource.com/
WSO2\textsuperscript{9} etc. Due to the necessary scope of this project, the evaluation is going to be based on some of these products/projects in order to accomplish the goal of the thesis.

\textsuperscript{7} http://www.mulesoft.org/
\textsuperscript{8} http://www.redhat.com/
\textsuperscript{9} http://wso2.com/
3. Problem Definition and Methodology

In the Literature Review, the background theory and basic concepts behind EAI, ESBs and MOM were discussed in details. The benefits offered by these approaches were presented as well, pointing out how they can facilitate in the application integration process. However, there are certain downsides that need to be discussed as well, which set the ground for the problem definition of this thesis. These downsides were taken into consideration when the Access Point Framework was developed, with the main goal of overcoming them as much as possible. The following section discusses these issues in greater details.

3.1. Problems of existing integration solutions

One of the most problematic issues in Enterprise Application Integration has been to find an efficient and scalable way to transform the messages in formats understandable by the receiving applications and transmit them from one application to the other. There are two basic topologies for application integration: hub-and-spoke and bus (Goel, 2006).

![Hub and Spoke Topology](http://www.poltman.com/en/technical-information/eai/topologies)

Figure 3: Hub and Spoke Topology

As it can be seen from Figure 3, there is a centralized hub which is responsible for transforming the messages from a format acceptable by one application to a format understandable by another application; the other responsibility of the hub is to also route the

---

messages from the incoming spoke to the destined application via its corresponding spoke. The role of the spokes, meanwhile, is to transform the application specific format to a format understandable by the hub (Goel, 2006). In that manner, three problems can be immediately identified with this topology:

- First, although this architecture “overcomes” the mess of point to point connection, what it actually does is putting all those connections into a centralized “black box” - the hub. This way, one single node has to take care of all the connections (e.g. maintaining routing tables) between the applications, which leads to the second problem.

- Putting all the processing of messages and routing into a single hub eventually creates a bottleneck as the traffic increases, as well as single point of failure (Tutorials.jenkov.com, 2012). Of course, having a more powerful machine doesn’t always solve the problem in practice (Goel, 2006).

- The third problem related to this topology, are the transformations which the hub is responsible for. The fact that the hub has to transform every message into format understandable by the receiving application creates an enormous overhead and consumes great amount of the node’s processing power (Microsoft Corporation, 2004).

Another approach is the Bus topology, presented in Figure 4.

![Bus Topology](image)

**Figure 4: Bus Topology**

The difference here is that between the bus and the application, there is an adapter which takes over the transformation to application understandable format, while the bus is
used only for relaying messages. The adapters subscribe to the bus and pick up the messages for the associated application. Usually, the adapter and the application reside on a single node which completely removes the processing overhead of the transformations from the bus (Goel, 2006).

A common way of implementing these two topologies can be illustrated with messaging services. Examples of the hub and spoke topology can be Message Brokers (MB) and Transaction Brokers (TB). An example of the bus topology is the Enterprise Service Bus (ESB). All of which will be responsible for the message routing, transformation, etc. Nevertheless, each of these technologies is limited in its capabilities, depending mainly on the vendor's implementation, and for certain companies that have a very high transaction volume, this becomes a serious issue. As potential solution for scalability of these technologies is adding more components, for example, adding more Message Brokers or additional ESBs in order to improve the transaction flow within the architecture. However, the downside to this solution is the pricing of these components, which means that by adding more components, the costs of the company for building the architecture will increase (Woolley, 2006).

Moreover, adding more components, does not always promise increased scalability. In case there is a necessity for large data blocks to be transmitted, this will slow down the communication. The ESBs or MBs will still have to handle, both, the rest of the messages and the transmission of large files which consumes a fair amount of their capacity. Therefore, adding more such components is not the most convenient way. Even if the components can be obtained out of charge, in the case of open source implementations, appropriate hardware is required on which these components will run (Shaheen, 2010).

These are some of the problems which still exist with the current implementations of ESBs and the Access Point Framework (APF), mentioned previously, offers potential solutions to the aforementioned issues, through a simple and thus elegant design. The solutions provided by this framework will be discussed throughout this thesis paper, by comparing them, at the same time, with existing implementations in order to point out the advantages which the framework offers.

### 3.2. Goals and Objectives

Having in mind the common issues of the application integration solutions, discussed in the previous section, the focus of this thesis will go in two directions and try to achieve the following two goals:

- Compare the APF with two existing ESB solutions and point out how this framework avoids some of their potential drawbacks;
- Integrate the APF with two open source MOMs and provide evaluation for both, that will help potential users to choose the more appropriate one, according to their specific needs
In order to achieve these two goals, the following objectives were set to assist the process:

- Choose two ESB implementations and investigate their architecture and functionality
- Compare the characteristics of the two ESBs in relation to the APF and discuss the advantages of the APF over them
- Choose two open source MOMs to be integrated with the APF
- Integrate them in the existing implementation
- Discuss their characteristics in comparison to each other
- Evaluate the performance of the two MOMs in comparison to each other
- Integrate two applications using the APF
- Evaluate the performance of the APF itself

Having summarized the goals and objectives of the thesis, in the next section the methodology devised to achieve them will be discussed.

3.3. Methodology

In order to reach the discussed objectives, certain methods have to be devised that will facilitate this process. This section concentrates on discussing the methods used for investigating the existing ESB and MOM implementations, the criteria used for comparing the chosen ESBs with the APF, the evaluated features of the selected MOMs, as well as the guiding points for evaluation of the chosen MOMs and the APF itself.

3.3.1. Research

The research conveyed as part of this project was of both qualitative and quantitative nature. The qualitative part focuses on answering questions such as why is the APF important and how does it provide potential solutions to the problems discussed in the problem definition; what are the benefits of the APF compared to the existing ESB products. Additionally, it provides in-depth analysis of how well do the chosen MOMs support the features discussed in section 3.3.7. The answers of these questions were produced through gathering knowledge from both primary and secondary sources. Primary, in terms of discussions with the supervisor, who is an expert in this area, on his past experiences with the technologies taken into consideration; secondary sources in terms of articles presenting similar work or documentation on the technologies in place. Having elaborated answers on these questions throughout the thesis constitutes an overview of the qualitative research done on these topics.

On the other hand, in the evaluation part of the project, quantitative research took place. The performance evaluation relied on measurements taken through benchmark tests, which were further used for calculation of certain criteria (discussed in sections 3.3.5 and 3.3.8),
which would give an overall picture regarding the performance for both the MOMs and the APF itself.

3.3.2. Investigating existing ESBs and MOMs

In order to find, first of all, two suitable ESB products, which will be compared, a preliminary study was conveyed to see what is offered on the market. For this purpose existing evaluations of products were reviewed, as well as some existing documentation which was available for them. The research discussed in the Literature Review under the section Related Work, was mainly what facilitated the investigation of the products.

In a similar manner, the MOM products were reviewed. However, in this case, besides researching the documentation and existing comparisons, we tried to take a look at some personal experiences with the available MOMs. This type of information was mainly gathered from personal blogs of experts in Message Oriented Middleware, as well as discussions regarding personal experiences with the technologies of the supervisor of the project. Of course, the main decision was not based only on these shared experiences, but they were found useful in the selection process of the two final MOMs.

Overall, in this manner, the available pool of technologies was carefully studied. The following sections on choosing the final products will cover in details the reasoning behind the choices made.

3.3.3. Choosing the ESB products

For the purpose of achieving the first goal, two ESBs had to be selected for comparison with the APF. There is a wide number of existing ESB products that could be taken into consideration; however, due to the scope of the thesis, it was considered that two implementations will suffice for contrasting and emphasizing the benefits that the APF offers, when it comes to the problems discussed in the Problem Definition section. A prerequisite for the choice was that one product should be open source and the other should be commercial. Usually this is one of the key factors when people are choosing an ESB product, so it was decided to have one representative from each category.

Eventually, the decision was narrowed down to Mule ESB (the open source product) and Microsoft’s BizTalk ESB Toolkit (the commercial product). The main reason for choosing Mule ESB was the fact that, according to (Vollmer, 2011), this is one of the most commonly used open source solution, which at the same time is a very solid product. Also, Mule ESB is one of the most frequent ESBs that appear in the literature, when it comes to evaluation of such products.

On the other hand, the reason behind choosing BizTalk was slightly different. It was clear that this product was not one of the main players on the ESB market; however, we considered that the complexity of the implementation can serve as an example that truly reflects the problems identified with existing ESB products.
Having chosen the two products that will be compared with the APF, also the criteria for the comparison had to be chosen. They are presented in the following section.

3.3.4. Comparison Criteria for the ESBs

In order to compare properly the two chosen ESB implementations with the APF, a guiding set of criteria that will be evaluated, was devised to assist in the process. Below is a list of the criteria:

- Similarities/differences between the building components of the frameworks and their functionality
- “Plugability” aspect of the architecture (are the components easily separated from or connected to each other)
- Support for component distribution for load balancing
- Ability to push the transformations to the application side
- Flexibility in building different topologies with the frameworks
- Message format of the implementations
- Supported types of messages interchanged over the framework
- Persistence of messages
- Similarities/differences in the message flow within the frameworks
- Ability to integrate with message brokers
- Support for message filters and message routing
- Support for content based routing
- Availability of Graphical User Interface
- Use of configuration files
- Technologies used
- Lock-in to a specific vendor when it comes to choice of underlying technologies

Having the complete list of topics that will be covered in the comparison, it is important to mention that not all criteria will be used for both comparisons. Some of them are more specific for one ESB product rather than the other. Therefore, although the majority of the criteria will overlap for both Mule ESB and BizTalk ESB Toolkit, some specific ones, might be found in one and not in the other and vice versa.

3.3.5. Evaluating the APF

In order to evaluate the performance of the APF, regardless of the chosen message queues, another set of measurements were taken. Below is a list of the evaluated aspects:

- Time needed for a message to pass through each of the APF’s components separately
- Time needed for a message to pass through an APF instance as a whole, forward and backwards
- Number of APF interactions per second
These criteria reflect the performance of the APF’s current implementation, however there are certain implications on the results obtained. The reasoning and elaborated details can be also found in Chapter 7.

Up until this point, the discussion was oriented towards the methodology for achieving the first goal. The next three sections discuss the details behind the process for reaching the second goal.

3.3.6. Choosing the MOMs

This section discusses the process of choosing two MOMs that will be integrated with the APF to enable the messaging backbone for the communicating applications. One of the prerequisites was to choose an open source solution in order to avoid the possible development costs, which helped in narrowing down the choice. Moreover, from the literature review, two important realizations were made, which created additional prerequisites for the solutions to be chosen. First of all, it is important for at least one of the choices to be based on the JMS specification, since it is a well-established standard, on which the majority of the MOMs adhere to (Curry, 2004; Chappell, 2004). The other prerequisite is related to the problem of interoperability among MOM solutions. As explained in the literature review, an emerging protocol that aims towards interoperability between MOMs is AMQP, and therefore, it is a prerequisite also for one of the messaging solutions to implement this protocol.

Having all these considerations in mind, the final choice was narrowed to ActiveMQ and RabbitMQ. Some additional facts that helped in making the final choice for both MOMs are presented below:

**ActiveMQ:**
- This MOM is implemented in Java and has a Java client which offers an easy way to be integrated with a Java based framework, such as the APF;
- It adheres to JMS 1.1;
- Third reason for choosing this MOM was the fact that it has wide and active support community;
- Implemented by well-established software foundation – Apache;

**RabbitMQ:**
- Based on AMQP;
- It is implemented in Erlang, whose foundation is inter-process communication through message queues and offers good performance and scalability (GoogleTechTalks, 2008)
- Used by large organizations in some of their projects, like: VMware (in their cloud services and virtualization products), Google (in Rocksteady project), Mozilla (in Pulse project) (Amqp.org, 2012)
3.3.7. Feature evaluation for the MOMs

In order to critically discuss the two chosen MOMs, especially in comparison one to the other, a list of features was prepared, that are of main interest when looking for a suitable MOM. Each of the MOMs was evaluated according to the features, creating an overall picture that a potential user can find useful when choosing an appropriate MOM for the APF. The evaluation list is presented below:

- Available clients
- The size and activity of the Support community
- Communication protocols supported (such as TCP, UDP, etc.)
- Interoperability protocols supported (such as AMQP, OpenWire, STOMP)
- Support for SSL
- Conformance to the JMS standard
- Routing
- Clustering

It has to be mentioned that not all of this criteria might be of interest to a user. In different scenarios, some might be more important than the others. However, having all of them discussed in Chapter 6 subsection 6.2.1, creates a useful guide for the majority of future users, especially when combined with the evaluation from Chapter 7.

3.3.8. Evaluating the MOMs

In the process of evaluating the MOMs, the greatest focus was their performance in comparison one to the other. In order to see how well they perform, the evaluation would be done in two environments – on single machine and in distributed manner (on two machines) in both persistent and non-persistent modes. The measurements taken will reflect the following:

- Throughput
- Average Latency of the messages
- Time needed for transferring certain number of messages
- Throughput variation through the execution time for certain number of messages

Overall, the results from these measurements will clearly reflect which MOM will perform better in which situation. This will be an essential criterion that will facilitate the potential users of the APF to choose the appropriate MOM for their needs. The details of the performed evaluation can be found in Chapter 7.
4. The Access Point Framework

The Access Point Framework (APF), is an Enterprise Application Integration Framework, based on message queues, for transforming application native messages into Enterprise Standard Platform Messages. The chapter starts with a detailed description of the APF components. This is followed by a presentation of the APF architecture, together with the different deployment models which it supports. The last section consists of a discussion about the benefits of the framework, in relation to the problem definition statement. The description of the components and the architecture were done based on the existing documentation of the APF within Aurenav LLC.

4.1. Components of the APF

The APF consists of the following three main components:

- **Application Integration Component (AIC)** – provides adapter for the application to connect to the APF
- **Transformation and Validation Engine (TVE)** – does transformations and validates the messages that pass through the APF
- **Messaging Platform Adapter Service (MPAS)** – is responsible for sending/receiving the messages on the framework

Besides these three, there are also some supporting components:

- **Base Components** - keeping the most commonly used classes such as service context, logger functionalities, JMS utilities for sending/receiving messages, etc.
- **Repository Service (RS)** - responsible for keeping and setting the configurations for the main components
- **Access Point (AP)** – providing the Java classes for the components’ connectors and controllers
- **Message Warehouse (MW)** – responsible for storing the exchanged message requests and replies

In the following subsections the three main and four supporting components of the APF will be discussed in greater details.

4.1.1. Application Integration Component – AIC

The Application Integration Component (AIC) is the integrating component that comprises the APF framework. Using the functionalities offered by the APF requires an adaptation to the existing applications or services. There needs to be a common operation channel between the framework and the applications that need to be integrated. The adapter serves this functionality as shown through the literature review. Facilitating this requirement,
the AIC component offers an easy API that needs to be extended and modified so that the application that needs to be integrated would be able to communicate with the APF (Aurenav LLC, 2008a). However, it is important to note that an already enabled interface from the application or service side would require minimalistic AIC implementation. The AIC is an intermediary step required so the application would be able to establish a common entry point to the APF. The main functionality would be delivering messages to and from other applications.

The AIC component can be perceived as a mechanism used by the application to be able to receive notification upon messages from the APF but also for sending out messages when necessary through the APF. The AIC is a configurable component having all the configuration information defined within a single XML configuration file. The configuration file itself, contains information regarding connecting points of this particular component. This includes adapter information about where the messages come from (which application) and also the connection point for messages coming from another APF component, intended for the connected application (Aurenav LLC, 2008c). One other important configuration information is regarding the processing steps that are required by the AIC in order to facilitate the integration process.

The processing steps, more specifically actions, within the APF terminology, are defined in the configuration file as well. They can specify how the message that flows from the application to the following component would be processed internally within the AIC component. Here it should be noted that it is a two way process. Both received messages from another component and outgoing messages from the application are being processed by the action steps that have been defined. Once a message is within a component, the action steps are applied to the message so that it is ready for the following component. This offers great flexibility in adjusting the AIC component based on the requirements of the application and in general on the integration solution requirements (Baldwin, 2012). One important aspect to notice, as it will be seen later on, is that the AIC component is designed in such a manner to support extensibility easily. In other words, a number of action steps can be supported upon request.

For instance, if the component that follows the AIC uses Java objects but the calling application has only XML documents, the action steps can specify how the adaptation would take place before forwarding the message to the other component. At present time of writing, the AIC component supports this functionality, action that can be used for transforming XML to Java and vice versa. This way it enables adapting communication between applications that provide XML documents and Java native language that is used by the APF. These action steps can also be omitted if the applications that need to communicate do not need any processing steps within this component.

The code snippet below represents an actual configuration file from the APF framework.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<aicconfig>
```
<message>
  <actions>
    <JavaXmlAction>
      <ActionParameters>
        <JavaObjectXMLMappingParams>
          <objectFactoryName>com.aurenav.apf.aic.action.handler.java.object.JavaObjectFactory</objectFactoryName>
          <objectFactoryCreateMethodName>createObjectForMsg</objectFactoryCreateMethodName>
          <javaToXMLMethodName>getContentsAsXml</javaToXMLMethodName>
          <xmlToJavaMethodName>setContentsFromXml</xmlToJavaMethodName>
        </JavaObjectXMLMappingParams>
      </ActionParameters>
    </JavaXmlAction>
  </actions>
  <InConnector>
    <APIConnector>
      <ConnectorFactory>RSSAicAPIConnectorFactory</ConnectorFactory>
    </APIConnector>
  </InConnector>
  <OutConnector>
    <APIConnector>
      <ConnectorFactory>com.aurenav.apf.tve.connector.TveConnectorFactory</ConnectorFactory>
    </APIConnector>
  </OutConnector>
</message>

**Code Snippet 1: AIC configuration file**

### 4.1.2. Transformation and Validation Engine – TVE

Following a similar structure is the next component that is included within the APF framework, which is the Transformation and Validation Engine (TVE) component. As the message passes through the AIC component it typically proceeds to the TVE component. This is to represent a logical separation between the adapter concept and the transformation concepts identified in the literature review. In some cases it might not be mandatory for the AIC to be present. The application can extend the TVE component as well, having the TVE act as a common entry component through which the application communicates with the APF (Aurenav LLC, 2008d). In other cases, after the messages have passed through the AIC component, the TVE component assumes the responsibility of the transformation activities that are most typical for integration scenarios. The functionality is to convert any application specific information format into the format of the defined canonical model understandable by all recipients of the message (Aurenav LLC, 2008d).

In order to perform this functionality, the TVE component also comprises of a number of processing steps or actions that were previously mentioned. It too follows the pattern of a
configurable component containing all the necessary information within a single configuration file that is easy to modify. The configured file, typically, consists of number of action steps that are identified for the particular component and information about both the flow to and from the component (Aurenav LLC, 2008c). This means, which component it needs to receive messages from (the AIC for example) and the component that the messages have to be forwarded to (the MPAS that will be elaborated in the following section). The action steps can be applied in both ways depending on how it is needed.

The differentiating factor is within the action steps that are to be defined for the TVE. These define how the message is to be processed once within this particular component. As it will be shown in the design further on, it is designed with modifiability and extensibility in mind to support a number of configurations (Aurenav LLC, 2008d). This would mean, either including or excluding existing action steps or even implement additional functionality steps, as required by the specific integration requirements. The included action steps that are part of the TVE, are XSLT transformation, velocity transformation11, XSD validation on XML requests, but are not limited by only these. Others could be supported as well, such as security, cryptography, but also business actions such as applying rule actions (Aurenav LLC, 2008d).

The included action steps are common among integration solutions and are required for establishing a common communication “language” among applications, as identified in the literature review. An example would be when an application sends a particular XML document that has been converted by the AIC component and it can be then processed by the TVE component. The TVE then applies the transformation and validation actions based on the configuration information, making the message ready to be forwarded to the component that follows.

This way, for instance, the message coming from an application would typically be transformed based on the XSLT transformation actions to the canonical message model (CMM) of the APF. Afterwards, forwarded to the MPAS component described later on. A message that is coming from the APF, needs also to be transformed and validated from the very same CMM to the application specific format. Again this is based on the XSLT actions for incoming messages that have been defined in the configuration file.

4.1.3. Messaging Platform Adapter Service – MPAS

Last but not least, from the main components of the APF, there is the Messaging Platform Adapter Service (MPAS) component. Covering the first two components that messages would typically pass through, the message is forwarded to the MPAS component. The component represents another logical separation that is part of the integration solution, that of loosely coupling the applications. The main technology enabling the loose coupling is a message queuing solution that provides asynchronous communication between applications.

11 http://velocity.apache.org/
The functional description is to govern the interchange of messages between applications on the communication channels (2008c, Aurenav LLC). Additional supporting functionality includes handling internal APF maintenance messages and error logging messages through the use of queues. Last, for integrity purposes all incoming and outgoing messages are being stored within a Message Warehouse, another APF component that is elaborated later on. The MPAS represents the final step of actually forwarding the messages from one application to another and representing the infrastructure for the connection between the APF components on disparate machines.

Having grasped the main concept around the MPAS component, it can be said that the component is configurable and that the information for the configuration is contained in a single configuration file. This component can also include actions that are specific to the processing of the message required by the integration solution. For instance message header information that could be appended to the message before sending it over the queues (2008c, Aurenav LLC). However, at the moment of writing the MPAS, no particular actions were explicitly included in the configuration file. The actions are part of the developed solution. This does not exclude or limit the design and the capability of the MPAS, since it, too, offers the possibility to be extended. Based on future requirements a number of actions could be specified and implemented.

The connection points of the MPAS component are configurable. Typically, one connection point is for the component that would be sending messages to the MPAS component. The other connection point is for the messages from the MPAS to the channels used for exchanging messages (Aurenav LLC, 2008c). The channels are where the notable difference from the other components can be identified through the use of queues. Thus, there are four main channels that have been identified where the MPAS could possibly route the message, presented in Figure 5. Each application needs to maintain four queues:

- IN Queue
- OUT Queue
- ERROR Queue
- MAINTENANCE Queue

Starting from the beginning, the first two queues enable the two way communication from and to the application that needs to be integrated. In that manner, the IN Queue is where all incoming messages are being received and the OUT Queue is the channel used by the MPAS for sending the messages from the application. The ERROR Queue is intended for handing messages that have been issued by the communicating applications in case there are issues that need to be addressed with a particular application. Last but not least, the MAINTENANCE Queue exists for the purpose of internal maintenance of the APF solution that could possibly be used for updating template information in the Repository Server (explained later on) (Aurenav LLC, 2008c).
Besides the identified queues, the MPAS component is responsible for adding the message header before sending out the message. This too is a two way street, because the header needs to be extracted when receiving a message from another application. The message header contains information about the communicating services i.e. applications. The information consists of the Service, Locality, ServiceContext and Message classes. This information from the header is used later on the by the Message Warehouse for keeping track of the exchange of messages between services (Baldwin, 2012).

This leads to the last functionality of the MPAS component. After a message has been sent to the appropriate communication channel, it is stored for integrity purposes. The MPAS component sends a write request to the Message Warehouse component on each outgoing and incoming message (Aurenav LLC, 2008c). The process and the details of the Message Warehouse are elaborated later in this chapter.

4.1.4. Repository Service – RS

Besides the three main components that were described in the previous sections, there are a number of supporting ones. Their functionality is to combine the internal APF components. The supporting components act and can easily be related to an engine responsible for ensuring that the APF is operational. One of those components is the Repository Service. Its main functionality includes reading configuration information necessary for the individual components of the APF (Baldwin, 2012). This includes, but is not limited to, information for the actions and connecting points for each of them. Besides the given functionality, it also represents the actual and logical storage point of the configuration information. It contains metadata about the location of the configuration files and the associated individual configuration files (Aurenav LLC, 2008c).

---

12 Image taken from: (Aurenav LLC, 2008c)
Upon a request from the APF, the RS has the functionality to provide the configuration information. It does so by trying to retrieve the information based on the configured paths to the correct configuration files for each of the components. Before extracting the information and returning it back, based on the request by the APF it could also check whether the information is stored in the Cache of the APF. Whenever a request is received by the APF from an application or another APF component, it can dynamically configure the component information. This is done either through the cache information stored or actually reading from the configuration files (Baldwin, 2012). When the configuration is read from the files it is also cached so that it can be used later on.

The extensible design allows for modifying the service in how the configurations’ information can be stored. This includes changes so that the information would be stored within a database or an EJB resource even retrieved from a message queue. At the time of writing, the APF supports file configuration that is modifiable and extensible. This functionality is extended with the capability of storing parts of the information within the cache, rather than just reading the information from a configuration file.

4.1.5. Base Components

Proceeding with other auxiliary components, there can be identified a number of basic components that are used by the APF. These serve as common base concepts surrounding the APF that enables the extensibility and modifiability. They are in a manner required for reusing commonly used functionality by the APF components. This includes the definition of the cache information, the Message format of the APF, exception framework for debugging purposes, logging utilities and JMS utilities used for the actual sending/receiving of messages (Aurenav LLC, 2008b).

The Base components represent, conceptually, aspects of the supporting functionality of the framework, i.e it constitutes the components that make messages move from one component to the following. It, too, is configurable in order to specify how the information can be retrieved so that the APF can be initialized in a proper manner (Baldwin, 2012). That would mean that the base components encompass the configuration for the services that would be used by the APF. Thus it entails metadata information regarding the whole APF instance that, as mentioned previously, could be used for logging purposes. Once the information is read from the configuration files, the cache could be used to run the APF instance environment which means faster retrieval of information. However, this component is not the only necessary for making the APF operational. The AP, that will be described in the section below, uses the classes defined within the base components for certain aspects as it will be seen.

4.1.6. Access Point – AP

The access point is yet another supporting component of the APF. It assumes the responsibility of defining other conceptually reusable components. The AP grasps the concepts that are necessary for enabling the integration solution. Afterwards, depending on
the necessary configuration, it allows for easy modification. The two main concepts that it entails are the component and the controller. In addition to these, it includes the action and the message header functionality (Baldwin, 2012). The component is designed to represent the actual components that were discussed, such as the AIC, TVE and MPAS and all accompanying configuration properties, such as the actions and connectors. Thus, the AP is supplementary standardized way of representing the components used by the APF itself.

In relation to the information based on the configurations that are defined in the individual files for each component, the AP is responsible to fetch that information from the configuration and initialize the component properly. This means that the AP should and does communicate with the Base components described previously and also the Repository Service which was covered in previous sections. The AP uses the Base components to check the validity of information regarding the APF instance, handle any exceptions that have happened through the Exception framework within the Base components and log the message flow. Thus, the AP, in collaboration with other supporting APF components, actually serves as an engine that keeps the message flowing among components. The Base serves as a supporting component for the APF since that is where a part of the core concepts and the cache are represented. The Repository Service, as it was described, serves another role within the complete framework of fetching information from the configuration files.

4.1.7. Message Warehouse – MW

Continuing forward to the last component, there is the Message Warehouse. The purpose of the component with the APF is to mark and preserve transactional integrity (Baldwin, 2012). This is achieved by storing each transaction within a database in order to track down if any message is lost during transactions. All messages that pass through to the MPAS component within the APF, after being send to the correct channel, the MPAS component proceeds with recording them to a local database (Figure 6). Each side has its own Message Warehouse database for this purpose. An outgoing message is recorded with a field marked as having an open status (O). After receiving a response back on that message, the MPAS forwards the message to the application and records, or better said, closes the opened status on the request message and stores the reply. This way, the information within the databases could be used to monitor the transactions that have occurred, both successful and failed, through the use of reports (Aurenav LLC, 2008c).

Figure 6: Message Warehouse

34
4.2. APF Architecture

Having discussed each of the components separately, in this section the main point of focus will be presenting how the components are organized in composing the whole architecture of the framework. The architecture can be clearly seen on Figure 7 below, where the high-level structure of the APF is shown:

As it can be seen on the picture above, the three main components – AIC, TVE and MPAS are clearly distinguished and they all have access to the Repository Service (RS).

Starting from the AIC, currently, regular Java calls can be used for the Application to connect to this component and send messages to the APF. Accordingly, XML to Java or vice versa transformations can be done within the AIC_Handler, if necessary, and further, this component will pass the message to the TVE. In order for the AIC to connect to the TVE, a correspondent TVE_Connector needs to be defined within the AIC’s configuration file. On a conceptual level, there are three possibilities for the AIC to connect to the TVE. One option is to do this with regular Java calls from the AIC to the TVE. Another option is to connect these two components through local message queues, which in Figure 7 is presented as JMS connectivity. And last but not least is to connect them through EJBs. In the current version of the APF, only Java calls are supported, and the JMS and EJB connectors are not implemented. However, by implementing this functionality in some of the future versions of the framework, actual physical separation of the components will be available, as it was intended in the design of the APF.

Image taken from: (Baldwin, 2008)
Furthermore, when the message reaches the TVE component, through the incoming connector (Java call, JMS or EJB), it takes care of the transformations and message validations necessary for the message. For the details regarding the possible actions within the TVE, refer to the dedicated section to this component. After the transformations and validations are completed, the TVE is supposed to send the message over to the MPAS component. For that purpose there has to be an outgoing MPAS_Connector defined within the TVE configuration file, pointing to the MPAS Connector Factory, which is responsible for instantiating this connector. After having the Connector instantiated, it can be used for accessing the MPAS_API through regular Java calls. For clarity purposes it has to be noted again that the JMS and EJB connectors between the TVE and MPAS are not implemented in this version.

In the lead, when the message reaches the MPAS, this component is responsible for putting it on one of the four existing message queues that can connect one APF instance with another. Therefore the MPAS will send the message off to another Application by placing it on the Outgoing Queue.

Another scenario is when an Application needs to receive this message. For that purpose, the MPAS is listening for incoming messages on the Incoming Queue. In case a message is received, it has to be propagated back to the Application, meaning from the MPAS to the TVE, where again transformations can be done to convert the message in the structure that the Application expects. The TVE, now will connect to the AIC which, in turn, will convert the message (if needed) from XML to Java Object, that is finally passed to the receiving Application.

Throughout this whole process, as it can be seen from Figure 7, the AIC, TVE and MPAS, need to have access to the Repository Service (RS) component. Within RS, as explained previously, all the configuration files, necessary for instantiating the components, are maintained. Accordingly, in the current version of the APF, the main components can access the RS through Java calls, but in one of the future implementations, this should be possible also through JMS and EJBs.

In that manner, it can be said that this was a brief overview of the architecture of the APF and the message flow going in both directions (starting from the application and going through the AIC, TVE and MPAS respectively, as well as returning a message back to the application through the components, in opposite direction). After having a basic idea of how the APF is organized, the different deployment models of this framework will be presented.

The first deployment model, presented in Figure 8 is the **Centralized Integration** model, where the Application and the Access Point (all the APF components) reside on a single host. This is the simplest architectural approach and is most suitable for low to medium bandwidth and processing overhead scenarios (Baldwin, 2008).
The second deployment model is **Distributed Integration**, with **EAI Approach**, presented on Figure 9. In this model, the Application and the AIC component, reside on one host (at the Application’s side), while the TVE and MPAS are located on the central server, which will provide these functionalities as an EAI Service. The benefit of this approach is the fact that by having these components on the central server, they would be easier to manage. Additionally, the approach supports linear scaling, as well as clustering and it is optimal for medium to high bandwidth and processing overhead scenarios (Baldwin, 2008).

**Figure 9: Distributed Integration; EAI Approach**

Last, but not least, is the second approach of Distributed Integration, which is the **Fully Distributed Approach**, on Figure 10. In this scenario, each component can reside on a separate host, which would enable linear scalability of the infrastructure, allowing the throughput to increase by adding more servers (Baldwin, 2008). Additionally, clustering is supported here as well, in order to avoid bottlenecks, in case one of the components has much bigger overhead than the rest. For example, if there is high transformation overhead, the TVE component can be clustered, so the two instances of TVE can share the processing demand.
This deployment model is suitable mostly for high to extremely high bandwidth and processing overhead scenarios.

Figure 10: Fully Distributed Integration\textsuperscript{13}

On Figure 11 is presented an integration of three applications, having the different deployment models functioning together. As it can be seen, the components can be separated physically in different manners, without affecting the actual integration process. The Different Access Points will still be able to communicate with each other through the central Message Oriented Middleware (MOM), regardless of the location of the rest of the components, (Baldwin, 2008). Accordingly, each Scenario has a short description provided in Table 1.

Figure 11: Deployment Models functioning together\textsuperscript{13}
4.3. Discussion

The APF facilitates the integration of applications and services throughout the Enterprise, regardless of the technologies in which they are implemented (Java, .Net, C++, etc.). Ideally, the concept of the APF is supposed to eliminate point to point integration between the applications, through the notion of routing. Although, routing is not supported in the current version, by implementing it in one of the future releases, it would overcome the need of connecting directly the applications one with another.

Besides conceptually overcoming the problem of point to point integration, the APF addresses also the second problem of the hub and spoke topology, discussed in the Problem Definition. As it is elaborated, routing does not solve the problem completely, since it is done centrally by a broker, and as the traffic increases, it is more difficult for the broker to cope with it. This can result in slowing down the message interchange or even crashing of the broker, considering that it is a single point of failure.

The way in which the APF offers solution to this, is by allowing clustering of its components, or more specifically, in this case, the MPAS component. This way, the broker can share the load with several lightweight MPAS components. For example, in case there is a huge load of traffic between two applications, additional dedicated MPAS components can be added, that can communicate directly, without having to pass the traffic through the broker (Zeromq.org, 2011). This can be especially useful if there is consecutive exchange of large BLOBs between two applications. Therefore, instead of passing the BLOBs through the broker, there is dedicated link between the two applications, using extra MPAS components, to directly exchange the messages, surpassing the broker.

Furthermore, the third problem posed by the hub and spoke topology was the fact that the hub is responsible for the transformations, which causes even more overhead. This problem is solved through the APF by offering an architecture where the transformations are completely removed from the broker. There is a separate component – the Transformation and Validation Component (TVE), which is responsible for doing all the transformations on the messages. Due to the flexible architecture of the APF, discussed in more details in section 4.2, and the ability to logically and physically separate the different components, it is possible
to even place the TVE component on a separate machine. In case there is great overhead on the central server, the TVE can be also pushed to the application side, in order to reduce the server’s load. Of course, the APF always allows the architect to choose where to put the component, depending on the processing overhead, in order create a solution that would fit most optimally the whole infrastructure.

In the lead, the beauty of the APF solution for scalability, is that adding extra components does not bring the disadvantages, discussed in the Problem Definition section, that usually come with adding extra components. One of the most common problems was the fact that extra components bring extra costs, which in this case is avoided considering the fact that the components can be clustered as much as necessary without any charges. On the other hand, the second problem mentioned was that more components usually means more processing power, or in cases even extra hardware, which again introduces cost. The way that the APF avoids this is by having the components very simple, lightweight, and separable one from each other. In case an extra component is required, it is not necessary for the whole APF to be added together with it, as it is the case in some of the existing ESB products. This minimizes the needed processing power related to adding more components.

As discussed above, the APF can improve the performance of the infrastructure which uses a broker, by introducing additional dedicated links between applications which can take part of the load off the broker. Additionally, the APF moves the centralized transformation outside of the central hub, which results again in reducing the processing power in the central point. And of course, maintaining the components separable and lightweight makes their clustering easy and less power demanding. In accordance, it can be said that the simple and elegant design of the Access Point Framework offers stable and scalable solution to the common problems in the area of Enterprise Application Integration. Therefore, this gives a brief picture of how the APF deals with the serious existing issues when it comes to application integration.
5. Comparison with ESB Products

This chapter provides a comparison between the two chosen ESB products – Mule ESB and BizTalk ESB Toolkit, with the APF. For each product a brief overview of the architecture is provided, as well as the similarities and the differences with the APF.

5.1. Mule Architecture

In order to be able to compare Mule ESB and the APF, first of all, Mule’s architecture needs to be briefly described. In Figure 12, the main concepts of Mule ESB are presented, as well as their connection, in order to enable the integration of two applications, in this case. Below, short overview of each concept is provided.

Figure 12: Mule ESB Architecture

- **Application**- These are the actual applications that need to be integrated. They can be of any kind, for example, a legacy system, a Java application, C++ application or even another Mule instance (Rademakers & Dirksen, 2008)

---

Image taken from: (Rademakers & Dirksen, 2008)
- **Channel**- These components provide the means for different external applications to communicate with the Mule instance. The messages exchanged between the applications are sent over the channels in place (MuleSoft, Inc., 2010). Optionally, channels can be used to connect different services together within the Mule instance (Rademakers & Dirksen, 2008);

- **Message Receiver**- It is responsible for receiving the data from the channel. Considering the fact that Mule supports different channel types such as HTTP channel, JMS, FTP, etc. (MuleSoft, Inc., 2010), correspondent message receivers are provided as well, which support these technologies, and are able to read the data accordingly;

- **Connector**- The previously mentioned message receiver is part of the connector, together with the message dispatcher and transformers. These 3 concepts form the connector, whose responsibility is to allow sending/receiving data through a specific protocol (Delia & Borg, 2008);

- **Transformer**- It is responsible for transforming data from one format to another, like for example from a CSV message into XML message, since different systems operate on different formats (Rademakers & Dirksen, 2008);

- **Inbound Router**- Once the message reaches the inbound router coming from the endpoint, the router regulates where this message will be sent to, or more specifically, to which service (Rademakers & Dirksen, 2008). Occasionally, several messages need to be aggregated into one, or the message needs to be filtered before sending it to the service component (Delia & Borg, 2008). These functionalities are responsibilities of the inbound router;

- **Component**- Within this service component, resides the implementation of the integration logic and it is usually a POJO, REST service, BPM etc. (Rademakers & Dirksen, 2008) For example a component which receives and invoice in the message can query its database and fill in some additional information to the invoice before sending it out again (MuleSoft, Inc., 2010);

- **Outbound Router**- Similarly to the inbound router, it specifies where the message processed by the service component will be routed to and passes it to the appropriate endpoint (Rademakers & Dirksen, 2008), (Delia & Borg, 2008).

- **Message Dispatcher**- Again similarly to the message receiver, this entity is responsible for sending the messages over the communication channel to the receiving application/service (Rademakers & Dirksen, 2008).

Correspondently, having an overview of the basic concepts of the architecture, a simple scenario can be presented of integrating two applications with Mule ESB, explaining how do the messages travel from the sending application to the receiving one.

42
As it can be seen from Figure 13, the sending application is responsible for sending an order to the receiving application. The order is sent as an XML message, due to the fact that this is the format which the sending application supports, and it is sent over an HTTP channel to the incoming endpoint which is http://myfirm.com/orders. In case a different communication channel is used, a different endpoint would be used as well, according to the transportation protocol, as discussed further in the scenario.

This endpoint is defined in the inbound router configuration, together with the correspondent transformer to which it will provide the message for the necessary transformation.

When the endpoint defined as http://myfirm.com/orders receives the XML message, it passes it to the transformer which transforms XML messages to Java Objects. This is necessary due to the fact that the targeted service (Customer Data Service Component),

---

15 Image taken from: (MuleSoft, Inc., 2010)
which will receive the message, accepts only Java Objects. Therefore, the transformer will take the information from the XML message and create a Java Object with that information.

When the transformation is completed, the Java Object is passed to the service or the Customer Data Service Component which queries its database to update the current information in the Java Object (for example add some details missing from the order), and through the outbound router sends it to a new endpoint which in this case is http://myfirm.com/verify. Due to the fact that the Verification service, to which this endpoint is connected, also operates with Java Objects, additional transformation is not necessary.

The inbound router of the verification service, similarly, contains the configuration for the endpoint on which the Java object is expected to arrive. When the message, or more specifically the Java Object, reaches this endpoint (http://myfirm.com/verify), via the inbound router is sent to the Inventory Verification Service Component. This service updates the order again with some extra information, such as the warehouse where the ordered items are located, and directs it to the correspondent outbound router. This, in turn, sends it to its correspondent endpoint, which in this case is jms://myqueue. Accordingly, this endpoint dispatches it on a correspondent JMS transport and the Order Fulfillment application receives it on the other end of the queue. Moreover, transformation is not necessary here as well since the receiving application operates with Java Objects.

From the example above it can be seen how Mule ESB and the correspondent components can be used to integrate two applications together. Now, having as a baseline the components and operation of both the APF and Mule ESB, they will be used to describe the similarities and differences between the two.

5.1.1. Similarities with APF

Initially, the comparison of similarities will be done based on the basic functionalities of the main ESB components, discussed in the Literature Review, in terms of what both of these frameworks offer.

In that manner, can be said, that both frameworks offer the adapter/endpoint functionality. Within the APF, this is provided within the AIC (Application Integration Component), which is an interface between the application and the rest of the APF. As described previously, the AIC component accepts messages from the application and conveys them to the TVE component. Similarly, in the case of Mule ESB, this concept is provided by the endpoint which allows the application to connect to the channel and send messages to the Mule ESB. The channels, connectors, dispatchers and receivers facilitate the functionality of the endpoints (Rademakers & Dirksen, 2008).

Furthermore, a similarity can be found between the TVE (Transformation and Validation Engine) component of the APF and the transformer component of Mule ESB. They are both responsible for transforming the message into the necessary format. The TVE, besides message transformation (XSLT and Velocity), offers XSD validation on XML messages, security checks, cryptographic unit and definition of rules that perform application
specific functionality. The transformers of the Mule ESB are more narrowed to just doing a transformation from one format to another, for example from TextMessage to String (Rademakers & Dirksen, 2008). Of course, besides the readymade transformers offered by Mule, the users have the possibility to define their own ones (Delia & Borg, 2008). The validation, on the other hand, in Mule can be done using filters, which are defined within the routers (MuleSoft, Inc., 2011).

A specific similarity regarding the transformation components of both frameworks is the fact that they are not mandatory as part of the architecture. The APF can function without a TVE component, having the AIC connecting directly to the MPAS component, in case transformations are not needed. In that manner the Mule ESB can also exclude transformers from its configuration, in case they are not necessary, as in the scenario discussed in the previous section (MuleSoft, Inc., 2010). From this can be said that both APF and Mule support the transformation of the incoming messages to a CMM (Canonical Message Model), but this step can be avoided if there is no explicit necessity for it.

In the lead, the third similar component type is the MPAS from the APF, in comparison to the inbound/outbound routers from Mule ESB. The MPAS has the responsibility of putting the messages on one of the four message queues which are part of the MPAS (In Queue, Out Queue, Error Queue and Maintenance) and receiving messages from these queues. Similarly in the case of Mule, the inbound and outbound routers also take care of the message routing. They receive the message on a certain inbound point, specify the actions to be done to the message (such as aggregating it with other messages or splitting it) and define where to further send the message (Delia & Borg, 2008).

Besides the similarity in the components, there is also similarity in the fact that both Mule ESB and the APF have a specific format of the messages local to the framework. Mule has a MuleMessage format, keeping the message in a Java Object, also some properties, attachments, and the MuleContext. On the other hand, the APF has its message format defined as Java Object called Message, keeping the actual message in a Java Object as well, together with the message type, correspondent business event, the name of the correspondent service and id. So overall, although they have different fields, the idea of keeping a single format within the framework is kept by both implementations.

Furthermore, another similarity for these two implementations, which is a major plus for both, compared to some other ESB implementations, is that they don’t lock you to a specific vendor when it comes to an underlying messaging or application server (MuleSoft, Inc., 2010). Any messaging solution can be integrated with the frameworks. This can even be seen in practice in the case of the APF. The initial implementation for DnB Nord bank was done using a commercial solution – WebSphere MQ, and the current implementation is with two open source solutions – ActiveMQ and RabbitMQ. On the other hand, as it can be seen in the case of Microsoft’s BizTalk implementation, locking the implementation to vendor specific solution is exactly what happens. Therefore, having this option open, as in the case of the APF and Mule ESB, gives the enterprise more freedom to choose an application or messaging server that will suit their needs accordingly.
In addition, both Mule ESB and the APF offer the benefit of creating different topologies with the existing instances. As it can be seen on Figure 14, there are several Mule ESB instances which can be connected in different ways to form different topologies like pipeline, peer network, hub-and-spoke, etc. depending on the necessities of the enterprise (MuleSoft, Inc., 2010). More specifically on the following figure, several topologies are combined in one, producing Enterprise Service Network, which is a topology combined of several others (Delia & Borg, 2008).

![Figure 14: Enterprise Service Network Topology with Mule](image)

Similarly, for the APF, on Figure 15 there is a specific topology implemented, in this case the ESB topology.
This can be easily modified if the APF instances are connected in different manner like for example pipeline which can be seen on Figure 16. A message is generated from Application 3 and sent to Application 1. Then Application 1 sends a message to Application 2, which receives this message and after the appropriate processing sends a message to Application 4. This way the applications are connected in a pipeline manner (the previous application feeds a message to the next one in line).

Figure 16: APF Pipeline Topology
Overall, in this section, the most important similarities between the two frameworks were covered. In the following section it is discussed how these frameworks differentiate from each other, emphasizing at the same time how these differences are beneficial for the one or the other.

### 5.1.2. Differences with APF

Besides the similarities between the two frameworks, there are noticeable differences as well which can make them suitable in different cases, depending on the requirements of the enterprise.

The first and foremost difference between Mule ESB and the APF is the connection between the components within the two. In the case of Mule ESB, the endpoints, transformers and inbound/outbound routers are tightly coupled one to the other. The configuration for all the components and their relationships is defined within one xml configuration file. An extract from such a file can be seen below:

```xml
<service name="Greeter">
  <inbound>
    <inbound-endpoint
      address="http://localhost:8080"
      transformer-refts="HttpRequestToNameString"
      synchronous="true"/>
  </inbound-endpoint>
  <vm:inbound-endpoint path="greeter"
    transformer-refts="StringToNameString"
    synchronous="true"/>
</inbound>

<component
class="org.mule.samples.hello.Greeter"/>

<outbound>
  <filtering-router>
    <vm:outbound-endpoint path="chitchatter"/>
    <payload-type-filter expectedType=
      "org.mule.samples.hello.NameString"/>
  </filtering-router>
  <filtering-router>
    <vm:outbound-endpoint path="userErrorHandler"/>
    <payload-type-filter expectedType=
      "java.lang.Exception"/>
  </filtering-router>
</outbound>
```

**Code Snippet 2: Mule Configuration File**

---

16 Code snippet taken from: (Delia & Borg, 2008)
As it can be seen from the file excerpt, all the components are defined within a service tag. More specifically, if the scope is set to the inbound router, for example, it can be seen that within its tags the endpoints are defined, and even more, within the endpoint itself, the transformer to be used is referenced. So all of them are tightly coupled one to another, meaning they all have to reside on the same machine and share the same resources.

This is not the case with the APF. Considering the fact that all of the components are independent of each other, they do not have to be tightly coupled as in the previous case. Each component has its own configuration file and the only connection between the components is specifying the Factory classes responsible for creating the connecting components. Of course, in this definition there are no details of a specific transformer, like for example above HttpServletRequestToString. All the details of the specific transformations are defined as Actions in the TVE component without having the AIC or MPAS knowing anything about them. This makes the components completely independent one from the other. Therefore, the APF offers loosely coupled components which can run even on separate machines balancing the processing load, avoiding single point of failure and overcoming the need to have all the work done centrally, as in the case of Mule ESB.

Additionally, being able to logically separate the components in the case of the APF, allows adding more instances of each component, if necessary, to the APF framework. For example, if there are many transformations that need to take place, instead of using one TVE component, two or three components of this type can be added to the framework. The same goes for the MPAS or the AIC, if needed. This increases the scalability of the solution as the load of messages increases as well.

Moving on to other differences, it can be said that in the current implementations, Mule ESB has support for ObjectMessages, TextMessages, ByteMessages etc., meaning that it has the support for sending files or whole objects from one endpoint to another, besides the regular String messages (Delia & Borg, 2008). On the other hand, the APF in its current version has support only for TextMessages, however this can be a useful functionality added in one of the later versions.

Another noticeable difference between the APF and the Mule ESB can be noted when comes to Java-to-XML/ XML-to-Java transformations. In the case of Mule ESB this happens like any other transformation – there is a transformer dedicated for doing that, which can be referenced from within an endpoint. However, for the APF, in the current implementation, it is necessary to convert the message from Java object into an XML format, within the AIC component, before doing further transformations and sending it off to another application. This is very convenient since XML allows for validation and transformation of the messages in different formats. Correspondently, when the XML message is received on the other end and transformed into a format suitable for the second application, it undergoes one more transformation in the AIC component, from XML to Java Object.

Of course, it must be said that this is a suitable scenario when the receiving application is in Java. If this is not the case, the configuration file for the AIC component can be updated
accordingly in order to avoid these transformations, but currently they are maintained due to the convenience of having XML as one of the underlying technologies for integration. The actual difference between the current implementations of the two frameworks is the fact of having the Java-to-XML and vice versa transformation as part of the AIC in the case of APF, while this is regular transformer for Mule ESB.

Moreover, in the current version of the APF, there isn’t any notion of filters. Filters specify conditions that, if met, allow for the message to be routed to the appropriate service. Mule offers this functionality, and in a similar way as in the case of transformers, some default filters are offered, together with the possibility of creating customized filters (MuleSoft, Inc., 2010), (Delia & Borg, 2008). However, this will be implemented in the APF in a future version together with the routing functionality. From the research conveyed, it was found out that in order to implement the routing and filtering functionality, Apache Camel can be used in combination with one of the messaging solution – ActiveMQ (Activemq.apache.org, 2004). This can easily bring the routing and filtering functionality to the APF and bring it to the same level as Mule ESB.

Furthermore, another currently unsupported functionality for the APF, compared to what Mule offers is a Graphical User Interface. Mule ESB allows the developers to drag and drop components and connect them one to another building the integration structure in that manner. However, although the APF still doesn’t offer a visual designer, it has very clear and easy to understand and modify configuration files which discard the necessity of coding.

One of the differences between Mule ESB and the APF, which is an advantage for the APF, is the fact that it stores the exchanged messages in a message warehouse (the requests and replies between the applications). This functionality, as already discussed, is useful in case of a failed transaction or lost message. Having it stored in the database, even if it doesn’t reach the other application, it is still not completely lost and it can be inspected at any time. On the other hand, the closest instance to a message warehouse in Mule ESB is the Object Store. Besides its other functionalities, it can be used for persisting the messages exchanged within the ESB. In that case it is referred as Queue Store. There are two most common types of queue stores: the “default in-memory-queue store” (which can store the non-persistent messages in memory) and the “default persistent-queue-store” (which is a file based store), (MuleSoft, Inc., 2011).

Nevertheless, there is an essential difference between persisting the messages and storing them in a message warehouse. The APF, which uses ActiveMQ or RabbitMQ also offers the functionality to persist the messages to avoid losing them in case of service failure. However, even if the message is persistent, when the service consumes the message, it cannot be further inspected, since it is removed from memory. With the message warehouse, on the other hand, this is not an issue. Even if the message is removed from memory, it is still stored in the database and can be further inspected, in case something goes wrong.

In that manner, the discussion regarding the differences between the two frameworks can be concluded. As it can be seen from the elaborated points, there are some aspects which
one of the frameworks has and the other lacks, and vice versa. Depending on what the enterprise requires as functionality from an ESB or an EAI, it can measure which of the provided functionalities are more important and go with the solution that has better support for those specific requirements. In summary, the important points are presented in Table 2.

**Table 2: Summary of comparisons Mule ESB - APF**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similarities between the building components their functionality</td>
<td>Both have adapter, transformation(optional in both frameworks) and messaging component</td>
</tr>
<tr>
<td>“Plugability” aspect of the architecture (are the components easily separated from or connected to each other)</td>
<td>The APF allows for distribution of components where each can run on separate machine; they are easily separable and plugable to each other; the components of Mule ESB are not separable in this manner; however, connecting them to each other is a very simple process</td>
</tr>
<tr>
<td>Support for component distribution for load balancing</td>
<td>The APF can have several instances of a single component in order to share between them the load balance; this is not the case per se with Mule ESB</td>
</tr>
<tr>
<td>Ability to push the transformations to the application side</td>
<td>The APF offers the possibility to have the TVE component on the same machine where the application resides and separate from the MPAS and the broker; In the case of Mule ESB, the transformations reside on the Mule instance with the rest of the components together</td>
</tr>
<tr>
<td>Flexibility in building different topologies with the frameworks</td>
<td>Both support building variety of topologies</td>
</tr>
<tr>
<td>Message format of the implementations</td>
<td>Both have their framework specific message format (MuleMessage and Message), used within the frameworks</td>
</tr>
<tr>
<td>Supported types of messages interchanged over the framework</td>
<td>Mule ESB has wider support for message types such as Object, Text, ByteMessages; the APF currently supports only TextMessages</td>
</tr>
<tr>
<td>Persistence of messages</td>
<td>The APF has a Message Warehouse, where each message is stored, in case one is lost; Mule ESB has Queue Store for that purpose</td>
</tr>
<tr>
<td>Similarity in the message flow within the frameworks</td>
<td>The message flow of both Mule ESB and the APF follows very similar pattern</td>
</tr>
<tr>
<td>Ability to integrate with message brokers</td>
<td>Both are able to integrate with variety of message brokers</td>
</tr>
<tr>
<td>Support for message filters and message routing</td>
<td>Mule ESB offers both filters and routers; these functionalities are planned in a future version of the APF and are not currently supported</td>
</tr>
<tr>
<td>Support for content based routing</td>
<td>Mule ESB offers content based routing; the APF currently doesn’t have that support</td>
</tr>
<tr>
<td>Availability of Graphical User Interface</td>
<td>Mule ESB offers GUI; the APF doesn’t have yet</td>
</tr>
<tr>
<td>Use of configuration files</td>
<td>Both support the use of configuration files; Mule ESB has a single configuration file while the APF has separate one for each component</td>
</tr>
</tbody>
</table>
5.2. BizTalk® ESB Toolkit Architecture

The following section covers the details of BizTalk server offered by Microsoft that consists of a set of essential abstracting concepts that are necessary when building integration solutions. The chapter contains information about both BizTalk server and BizTalk ESB Toolkit since the two are tightly interrelated, as it will be shown and seen from the presented information. The first part will include detailed information regarding the BizTalk server and elaborate the main components that consist and make up the BizTalk server. This will be followed by the BizTalk ESB toolkit, which is build on top of some of the BizTalk server components, together with some additional components offered by the BizTalk ESB toolkit.

**Receive location:** represents the logical entry point that contains the configuration and the actual endpoint (URL) that is needed to receive messages and the associated data. In addition to the initial receiving, it also contains information for the receive adapter and the receive pipeline which will be explained in the following two paragraphs.

**Receive/Send adapter:** the receive adapter is responsible after fetching the data with the specified protocol, to create a BizTalk message that would contain the actual message. In addition, the adapter adds any associated metadata information regarding the message and sends it over to a messaging engine (MSDN Library, 2009). The adapter can be anything from File readers polling a directory, HTTP or JMS connection, each of which is supported by BizTalk server adapter framework.

**Receive/Send pipeline:** Since BizTalk mainly operates with XML documents, the format of the receiving data would have to be converted to an XML document. It is configured as a receiving pipeline that performs transformation on the incoming message, such as decoding, disassembly and conversion to an XML format (MSDN Library, 2004a). In addition, the mapping can actually be done in the send port as well but it is typically used in the orchestration which is elaborated further in the document (MSDN Library, 2004b). The BizTalk works internally only with XML documents, which means that any message that comes in has to be converted to an XML format (MSDN Library, 2010d).

The above components are connected with the help of the **messaging engine**, responsible for transferring the messages from the adapter to the pipeline for further processing (MSDN Library, 2009). The messaging engine is present throughout the runtime processes of a single BitTalk application doing the actual delivery and fetching the message to various components so that the message is processed in the defined business process. Responsibilities include delivering messages from one component to the next within the lifecycle of the message processing (MSDN Library, 2010c).

After the pipeline processes, the messaging engine publishes the message to the **MessageBox** (MSDN Library, 2010b). The MessageBox is a SQL Server database that stores all incoming and outgoing messages, by adding entries to the tables. That would include the messages, or message parts and additional properties that might be used for processing the message accordingly. This step is done through a publish/subscribe protocol in order to
achieve the desired asynchrony. The messages are picked up by the specific subscribers based on a set of predefined filters and the matching message context. Typically these include orchestrations or send ports that are subscribed to the MessageBox. This, in terms, constitutes part of the routing configuration that the MessageBox performs. The properties information that came along with the message from the pipeline, is temporarily stored within the MessageBox. The information is needed in order to place the message on the right queue i.e. publish it. The message can afterwards be consumed by the correct subscriber of that queue.

**Orchestrations:** The logic that the message will follow is implemented through an orchestration or several orchestrations that are bound to a specific receive and send adapter and consist of executable code. Here, the workflow is specified that would automate the business process of the company and is independent from the programming language. The orchestration is typically connected to a receiving port and a sending port in order to be able to communicate to other business processes.

Figure 17 can serve as a summary of the components that were defined in the paragraphs above, and are considered as artifacts, conceptually speaking, in the BizTalk environment:

![Figure 17: BizTalk® Server Architecture](http://msdn.microsoft.com/en-us/library/aa578560)

Besides the described components and concepts in the previous section, the BizTalk ESB Toolkit consists of the following:

**On/off ramps**

The On ramps are the entry points to the BizTalk ESB toolkit application, where it actually is an abstract representation that is build on top of the BizTalk server Receive port with the notable difference that it uses BizTalk ESB toolkit pipeline components. The same applies to Off ramps which allow to dynamically allocate different physical endpoints the message will be routed to (Flanders, 2010).

---

Itinerary

An itinerary is a notion used to represent the metadata that describes how the message will be forwarded throughout the “bus”. This can be either an XML document attached to the message or perhaps based upon the content of the message as well. Afterwards, the components within the BizTalk toolkit use the information to correctly guide the message. The other components include, if necessary, routing for further processing through orchestrations, apply transformations or where the message needs to leave the bus environment. Itinerary processing is actually made up of BizTalk pipeline components that actually parse the instructions and, accordingly, can dynamically set up transformation properties or end point properties depending on the need and the metadata information (Flanders, 2010).

Resolvers

The resolvers are used to do dynamic resolution of service metadata, which would include fetching information for further runtime processing such as configuring routing information or transformation information. This would mean that the resolver would need to fetch the itinerary information which is specified typically in a database and would use that information to dynamically set the message context properties that are needed. In addition, the resolver can have a specific policy that would define the rules and conditions for applying the specific policy. The policy typically contains rules about endpoint information or transport information (Flanders, 2010).

This is where the difference lies, where even though the on/off ramps need to be bound to a specific endpoint port, the intermediary step of the resolver can actually be the one determining where the message would be further on send out i.e. sent to the appropriate off ramp. This would also mean how to call the right service that would perform a transformation.

Adapter providers

Adapter providers are used to set the configuration properties of a sending adapter, meaning, reading the mapping and metadata from the ESB toolkit configuration properties to the sending adapter context properties. This typically includes setting up the endpoint on the sending adapter (described in the BizTalk server section) if it is not known in advance or if it is not supposed to be static (Flanders 2010).

Exception Management

Exception Management is a framework within the environment of the ESB toolkit for unifying the complete processes for managing exceptions that occur in the BizTalk Server environment (Flanders 2010). It also has an API that can be used for creation, publication and management from orchestration processes.
There are also some additional components offered by the environment, that will not be considered in greater detail due to the fact that they are exposed as services rather than actually being part of the ESB environment. These are the ESB services that are offered by Microsoft and that can be used to delegate some of the processing and assist the framework in deciding on the processing steps of the message, once they are accepted on one of the on-ramps. Figure 18 is a reference to the BizTalk ESB Toolkit architecture, where the components that were elaborated are presented and the overview can serve as a better point of view for how the components are related.

![BizTalk ESB Toolkit Architecture](http://msdn.microsoft.com/en-us/biztalk/dd876606.aspx)

Figure 18: BizTalk® ESB Toolkit

As it was previously mentioned, all the components and abstractions described, are built on top of the existing BizTalk server technology. The notable difference is the resolver that, based on the itinerary information, can setup dynamically the configuration for endpoints. By this abstraction new components or services can be added and configured relatively easy in comparison to having them each being created statically. A BizTalk application has to have a configuration for the receive and the send port. Moreover, even the on/off abstractions are build on top of the dynamic send/receive ports that are offered by the BizTalk server.

Figure 19 offers a summary of differences between the ESB toolkit and the BizTalk components that were described previously.

---

The following example would show a typical runtime execution and the steps a message would go through when running in a BizTalk ESB toolkit (MSDN Library, 2010a; Flanders, 2010).

After a message has been received by one of the On ramps to a BizTalk ESB application, it can be further processed based on the receive pipeline configuration. The On ramp pipeline configuration can decide upon which itinerary is going to be used or the itinerary will be already with the message. This step is done through the use of the building or custom build pipelines defined for the On ramp. In any case, the On ramp has to be bound to a specific receive port which is configured, based on the existing BizTalk server settings.

In the case that there is no itinerary, the resolver, that was explained in the previous section, determines dynamically information that is used by the BizTalk ESB toolkit. Other functionalities, that are optional, include deciding on the end point information, calling upon an orchestration or even knowing which map to use based on the message. Basically, the component that is configured within the pipeline knows which resolver to initiate. The initiated resolver can then lookup the necessary itinerary from an itinerary store from an SQL server database. The resolver can also be dynamic, in the sense that if a UDDI3 resolver is used then it can communicate with an UDDI service.

After the message has been received and has passed through the On ramp and the pipeline components, it can be published to the MessageBox. From there, after the additional processing, the Off ramp can pick up the message and continue with the sending of the message through the send port. This step can also contain filters in order to select which messages will pass through and that have been published by a receive port. Also, another resolver can be used in the off ramp that can actually determine the physical endpoint and the delivery of the message. This can be done through the use of rules that will define the how

19 Image taken from white paper (Flanders, 2010)
the message would be configured. The benefit here is that the policy can be changed and the itinerary, through the use of the resolver, will be able to pick up these changes without affecting other BizTalk components, or demand redeployment.

In addition, one way to avoid publishing the message in the MessageBox, that is part of the existing BizTalk Server functionalities, is through the use of itineraries. This is as a consequence of the transformation or orchestration that could be incorporated in the resolvers and itineraries that were previously described. Typically, the transformations and orchestrations can be defined as Web services, thus when a message is being processed through the pipeline, they would be called upon. The resolvers can also contain information for routing and dynamically setting up endpoint information. This way any changes do not affect the system drastically, only a different service can be exposed, which will be defined in the policy as previously mentioned.

5.2.1. Similarities among architectures

The following section contains information regarding the identified similarities between the two architectures; that of BizTalk Server and BizTalk ESB Toolkit and APF.

The Application Integration Component (AIC) component from the APF framework can be related to the adapters that are defined within the BizTalk server and the BizTalk ESB toolkit. The endpoint that is defined, is similar to the settings that are set for the receive and send port within the BizTalk server or the On and Off ramp in the ESB Toolkit. The functionality that the AIC offers, is to represent the actual connection between application and the APF framework. It contains information about the endpoint and defines how messages enter the APF framework.

Through the component, the application can also receive notifications about messages coming from other applications. The component is designed in such a manner that it can include other application specific integration demands in order to facilitate better connection between the application and the APF. This would mean that it can include classes that would perform transformations to a format demanded by the APF for internal processing. Moreover, it contains information about the connection to the next processing component within the APF.

TVE component can be related to the Pipeline component that is part of the receive and send port concepts within the BizTalk ESB toolkit and BizTalk server. More specifically the various resolvers, that can be included, can be related to the TVE component from the APF. The pipeline consists of information about the different transformations that can occur based on what is demanded and the settings of the component, routing information is included or even an orchestration is called upon.

Similarly, the Transformation and Validation Engine from the APF is the middleware component that is responsible for performing various transformations on the received message based on the requirements of the application. This would include XSLT transformations, XSD validations, velocity transformation all of which are optional and more
can be added based on requirements. This allows for flexibility when it comes to performing operations on messages, similarly to the Policies that are defined in the BizTalk ESB toolkit. Within the APF a number of actions can be included that perform the operations on the message, just as in the BizTalk ESB toolkit can include a number of resolvers and policies.

The last component of the APF is the Messaging Platform Adapter Service (MPAS). This component takes the responsibility of the routing, i.e. putting the messages on the right channel. Within, the APF there are four identified channels, as mentioned before. Based on the flow of the messages to and from the application, the messages are propagated to the right channel. All messages that need to go to another application are put on the outbound channel and vice versa. The two other channels are for error handling within the APF and for maintenance, respectively.

The component can be mapped to the send port that is performed within the BizTalk ESB Toolkit or BizTalk Server. This is related to the resolvers that were described previously, when based on either the content or a specific itinerary, the message is sent to the appropriate port. Either way, it is the final step and it represents the actual sending of the message from one application to the next.

The processing steps within the APF are included in separate configuration files, for each of the components. This is similar to the policies that are defined within the BizTalk ESB toolkit server and that actually define what rules they would follow when it comes to processing of the message. Each of the components can have separate processing steps i.e. it can be related to the resolvers that are attached to all the separate policies that are defined at various processing steps.

Another similarity is the flexibility that both solutions offer, when it comes to functionality. Both within the BizTalk and the APF you could, but do not necessarily have to, include mapping functionality, routing or orchestrations. Meaning that the transformation of the message does not have to be present and it can just be omitted so the process could continue with forwarding the message to another component or to the off ramp. There are a number of variations that offer greater flexibility.

Further on, there is a similarity, conceptually speaking, in the aspect that both the APF and the BizTalk server persist the messages in a database. As stated in the description of the BizTalk ESB toolkit, each message, after being processed by the adapter and pipeline in the receiving location, is being published in the MessageBox before they can be processed by the correct subscribers. The reason behind this is that it is built on top of the existing BizTalk server architecture. The APF is also keeping track of the messages and validity by storing them in a MessageWarehouse. After a message has been received by the corresponding application, and a confirmation has been received about a message, it is marked as closed in the MessageWarehouse, meaning it has successfully completed.

Both the APF and BizTalk, have specific format of messages within their own framework. As it was shown previously, before publishing the message to the MessageBox it needs to be converted to a BizTalk specific XML message. It too is surrounded by message
properties that are defined in the context of the message. Similarly, in the APF, the actual message is converted to a Java Object, which besides the actual contents of the incoming message, has additional properties information such as the service that it has as a destination, the ID of the service, service name, business event etc.

The APF and the BizTalk server offer a number of deployment topologies, as it can be seen on the image below. In order to scale up a solution, a BizTalk server which is a default host, can have one or more host instances. A host is a logical container for the artifacts that were described previously, like receive locations, orchestrations or adapter handlers (pipelines). These are components that were previously described and that define a host within a logical container. In order to scale out, a host instance can be deployed on one or more physical BizTalk server computers based on the defined host. This way if one host instance representing a particular host fails, the other host instances can handle the failure since they represent the same host. Only one instance of a specific host can exist on a single server. Figure 20 below, can serve as an example of a deployment scenario for both hosts and host instances, in order to achieve greater availability.

![Figure 20: BizTalk® Server Hosts and Host Instances](http://msdn.microsoft.com/en-us/library/aa577415.aspx)

This concludes the identification of the similarities between the two frameworks. The following section elaborates more regarding the differences among the architectures.

### 5.2.2. Differences with the APF

In this section, the differences between the BizTalk and APF will be presented.

One of the essential differences between the APF and the BizTalk ESB Toolkit is the message flow. For instance, if a message that does not have an itinerary in the BizTalk server, it will have to be added based on a resolver and the resolver framework. The processing is

---

based on that particular itinerary which may include a number of different steps. The components within the APF framework are not completely operating based on the header or itinerary information about the message. Rather, as it was explained each component has a sequence of steps that are executed and are predefined. However, the architecture of the APF allows for component distribution, so that based on the needs and processing power, it can be deployed accordingly.

Another difference is that the components within the APF framework as we saw could be deployed independently. The configuration file for each of them is independent from the rest. Each configuration file corresponding to a single component has a set of steps that the component will follow without the need to be tied up with the other two components. The concepts that they are built upon allows for distributing them on different machine but at the same time achieving an image of one system. A notable difference is that most of the processing that will be done by the Biztalk server requires a definition of an on and off ramp where in the APF such thing is not necessary. Each component could be accessed separately and configured in such a manner that it does not require a single entry point.

The notable difference here is that, eventhough, several host machines can be created with the BizTalk server, the environment that would be deployed just for running one functionality, such as transformation, is bigger in comparison to deploying the APF component independently. Each host machine and instance could contain one or more artifacts that were previously described. Further on, the MessageBox will be present, eventhough, it can be controlled from a central point. There is more similarity with the BizTalk ESB toolkit where the MessageBox could be avoided, as the processing of the message might be different.

The BizTalk server could use information within the message or the context in order to dynamically resolve the end points. This is, of course, based on certain policies that are defined and do not affect the application when they are changed. The APF, on the other hand, at the moment is not explicitly related to the message content and context. This means that it does not use that information in order to process the message accordingly. It, rather, depends on the steps that have been defined in the configuration files. BizTalk ESB toolkit offers a modeling tool designer in order to specify development of associations between service providers and service consumers.

The APF uses message queues, at the moment, instead of publish/subscribe protocol. It means that the message will be stored until it has been received by the correct recipient and this way it guarantees delivery. Having the message stored until it is consumed, represents another safety net for message transfers between components, besides the other where the message is stored in a MessageWarehouse i.e. a database. It should be noted that APF does not actually poll the database or interact directly for consuming them from there. Rather it uses the broker by the messaging solution for achieving this, the database acts as a logger utility for observing the traffic and interchange of messages. Nonetheless, it realizes the necessary asynchrony between applications that need to be integrated.
Many of the operational settings that are required by the APF are setup through the use of configuration files. This is different to some extend to the BizTalk ESB toolkit and server, in the sense that there everything is setup through the use of development environment, where the associations can be specified through drag and drop development. The process for setting up the APF requires changes to take place in the files that have been defined as configuration files.

Since it is based on the existing BizTalk server architecture the ESB toolkit is completely tied up to the BizTalk server. This is because the Toolkit actually uses the information that is defined within the BizTalk application server in order to process the messages in a specific order. For instance, in order to define a specific port that an on ramp could use, it has to be tied up to a specific application and port that have been defined in the BizTalk server environment. With the toolkit, a specific ESB application has been developed in the BizTalk server, that has already ports and web services defined that can be used by an itinerary. Tying up the itinerary information, especially the on ramp with a specific port that is part of BizTalk application, is how it is considered that the ESB has been setup.

This is not the case with the APF. The APF itself can connect to different products offered by different vendors or have custom built applications. The action steps that are defined can be called upon as needed offering greater flexibility and vendor neutrality. Eventhough that the BizTalk server and ESB toolkit might provide already existing adapters, the environment and the complexity that surrounds it, is considerable. This leads to one more difference - the APF could be deployed in several topologies depending on the need, whereas the BizTalk server environment can host multiple host instances but not multiple host instance of the same host.

Another difference between BizTalk server and APF is also that the one is using the database as an intermediary for transportation where messages are being published and received by their subscribers. The APF relies currently on a message broker, so a number of different topologies and configurations can be setup to be used. The BizTalk server actually has subscribers to the database that query the database in order to know when a new message has arrived. At the same time, this offers the benefit for the BizTalk server to also perform routing and apply filters to the messages. At the moment, the APF does not support a very flexible routing configuration. The configuration is mostly dependent on the queues and topics that are defined to be used in the MPAS component. The logic is not typically dependent on the contents of the message, rather, on the defined processing steps within each of the components’ configuration files. The APF uses queues at its core rather than the publish/subscribe mechanism, however it could be deployed in this manner.

The BizTalk server environment supports for setting up multiple databases based on how the applications have been deployed. But no matter how many MessageBox databases exist, there always has to be one central “master” MessageBox database which will govern the routing of the messages to the other (secondary) MessageBox databases. One benefit of having multiple databases is that each application on a particular host, can query its own
database queue, rather than having all the host applications that exist on a network query the single MessageBox database constantly. An example can be seen on Figure 21.

4 BizTalk Server, 1 SQL Server Topology

6 BizTalk Server, 3 SQL Server Topology

Figure 21: BizTalk® SQL Server Topologies

In respect to the similarities in topologies that were described in the previous section, the APF can be deployed in different topologies as well. However, it was mentioned that a single host instance can exist in a particular server; meaning two host instances cannot represent a single host on the same machine. This is not the case with the APF, as it could be concluded from the description. This means that a particular host instance or a receive pipeline could potentially be deployed twice. This can be used to scale up the solution on different computers or even on the same one but as a separate process.

Having said that, it should be noted that the BizTalk server is actually proprietary software, thus it demands licensing. This might make the process of deploying the server in a distributed environment more expensive. It makes it more difficult to setup the BizTalk environment in a desirable and flexible manner that would offer better scalability and distribution.

This concludes the identification of both similarities and differences between the two architectures at hand. The summary of the issues identified can be found in Table 3.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similarities between the building components and their functionality</td>
<td>Both have an adapter, transformation and messaging component</td>
</tr>
<tr>
<td>“Plugability” aspect of the architecture (are the components easily</td>
<td>The APF allows for distribution of components where each can run on separate machine; they are easily separable and pluggable to each other; the BizTalk ESB Toolkit and Server environment are not separable in this manner;</td>
</tr>
<tr>
<td>separated from or connected to each other)</td>
<td></td>
</tr>
<tr>
<td>Support for component distribution for load balancing</td>
<td>The APF can have several instances of a single component in order to share between them the load balance; Within the BizTalk ESB toolkit and server environments, several databases and host instances can be grouped to share the load, however adding more instances would be costly</td>
</tr>
<tr>
<td>Ability to push the transformations to the application side</td>
<td>The APF offers the possibility to have the TVE component on the same machine where the application resides and separate from the MPAS and the broker; The BizTalk ESB Toolkit has the transformations on the host with the rest of the components together</td>
</tr>
<tr>
<td>Flexibility in building different topologies with the frameworks</td>
<td>Both support building variety of topologies; pricing could pose as a problem with BizTalk</td>
</tr>
<tr>
<td>Message format of the implementations</td>
<td>Both have specific message format to their environment, however they are different to their own environment</td>
</tr>
<tr>
<td>Persistence of messages</td>
<td>The APF has a Message Warehouse, where each message is stored, in case one is lost; BizTalk ESB uses the MessageBox concept for purposes of delivering the messages to the right subscribers and ensuring delivery</td>
</tr>
<tr>
<td>Similarity in the message flow within the frameworks</td>
<td>Different in the sense that the BizTalk environment could dynamically configure message flow; APF has a set of predefined steps that are nonetheless powerful</td>
</tr>
<tr>
<td>Ability to integrate with message brokers</td>
<td>BizTalk ESB Toolkit uses underlying BizTalk Server concepts as a message broker but could be integrated with others through On/Off ramps; APF can integrate with a variety of brokers.</td>
</tr>
<tr>
<td>Support for message filters and message routing</td>
<td>BizTalk ESB Toolkit can dynamically resolve the next destination of a message based on certain itineraries and resolvers; these functionalities are currently not part of the current APF version</td>
</tr>
<tr>
<td>Availability of Graphical User Interface</td>
<td>BizTalk ESB toolkit offers a GUI for development; the APF doesn’t have GUI,</td>
</tr>
<tr>
<td>Use of configuration files</td>
<td>The itinerary information are in an XML file that is configurable; in the APF each component uses a separate configuration file</td>
</tr>
<tr>
<td>Technologies used</td>
<td>APF is build on open source solutions; BizTalk is a commercial product</td>
</tr>
</tbody>
</table>
Overall, this concludes the comparison chapter between the APF and the two chosen ESB products-Mule ESB and BizTalk ESB Toolkit. As it is visible from the information presented, the APF offers advantages and greater benefits when it comes to the separability of the components, their clustering and ability to deploy them in distributed manner, which is the road to increase in scalability. It is true that, currently, the two ESB products offer a wider palette of functionalities, however, through further development, the APF can certainly catch up as well.
6. Implementation

This chapter discusses the implementation work which was done throughout the thesis project. Although the main goals of this project did not require great amount of implementation, certain adjustments to the existing code of the APF had to be done in order to make it completely open-source. Additionally, in order to prove the functionality of this framework, two simple applications were implemented and integrated using the APF. Overall, the following three areas were the main implementation parts of the project:

- Replacing the existing Oracle Database (which was underlying technology of the Message Warehouse) with MySQL Database;
- Replacing WebSphere MQ with two open-source MOMs: ActiveMQ and RabbitMQ
- Implement and integrate two simple applications

The details behind the implementation of each of the three areas can be found in the correspondent section below.

6.1. Database Replacement

The replacement of the database was quite straightforward. Initially, MySQL server was installed, followed by creating the database which will represent the message warehouse. There were certain prerequisites that had to be followed when creating the database, such as securing it with a specific username and password. The next step was creating the two basic tables expected in the message warehouse. One table stores the complete Requests and all the details related to them. The other table stores the Replies. Having prepared the underlying database, the next step was altering the code.

The main code alterations consisted of replacing Oracle-specific Java statements into MySQL-specific ones. For example, instead of opening OracleConnection, creating OraclePreparedStatement, etc, these were replaced with MySQLConnection and simply PreparedStatement. Moreover, a similar situation was encountered in some of the datatypes of the values stored in the database. For storing a Blob object (which was the actual message to be saved), there is difference in the Java code when it comes to Oracle and MySQL. In that manner, certain adjustments had to be made.

Overall, it can be said that this was a brief overview of the work done in the replacement process of the database. The next section goes into the details of replacing the message queues.
6.2. Messaging Middleware Replacement

The integration process continued with the modification and integration of the two messaging solutions. Before proceeding with the implementation details, the two solutions will be presented through the set of features that were identified and presented in subsection 3.3.7. Afterwards, the section proceeds with the implementation details. For the purposes of this task, there was a learning curve that needed to be surpassed in order to perform the task. The first step was to integrate the ActiveMQ solution by modifying the configuration files and necessary properties as it will be seen. The second step was to integrate with another solution, which was RabbitMQ.

6.2.1. Feature Evaluation

6.2.1.1. ActiveMQ

ActiveMQ is an open source messaging and integration patterns server, which has been widely deployed and used. It is one of the many Apache Software Foundation projects offered by the Apache Software Foundation\(^22\) most notably known for its HTTP server. This development community is great counting more than 50,000\(^22\) members and besides being known as a web server, it also has a number of open source enterprise level projects, among which ActiveMQ is one that is most interesting for the purposes of this work. The community support was an essential aspect that governed the choice behind this open source solution.

ActiveMQ is a messaging system that is Java based and conforming to JMS 1.1 having a large number of supporting clients for C, C++, C#, .NET, Erlang, Perl and many others. In addition, they do provide their own protocol for interoperability named OpenWire, but also STOMP support. The OpenWire protocol is a binary protocol aimed at achieving higher performance, having clients in Java, C, C++ and C#. The protocol does not solve all problems in relation to interoperability since it is specific to the ActiveMQ messaging system. ActiveMQ is considered as the reference implementation of also STOMP. The protocol is considered as a simple wire protocol, which is text based and not as inclined to high performance and reliability as the based protocols, AMQP but also OpenWire\(^23,24\). At the time of writing, the AMQP was not supported, however, the protocol is supported by another Apache project, named QPID, which is not as mature as ActiveMQ.

ActiveMQ offers a number of features which are well establish in the MOM world. It goes without saying that both point-to-point and publish-subscribe messaging models are supported, with persistence modes and acknowledgements. The persistence is supported by a JDBC connection having the ability to switch from the default database provided with the solution. It is also designed for creating server clusters that represent a single logical server, but also peer to peer and client-server communication. In addition, there are modules for establishing communication through a number of protocols such TCP, SSL, NIO, UDP,

\(^22\) [http://www.apache.org/](http://www.apache.org/)
multicast, JGroups and JXTA transports. Another key advantage is that it has been tested in a number of J2EE servers such as Geronimo, JBoss 4, GlassFish and WebLogic\textsuperscript{25}. The open source Mule ESB solution utilizes ActiveMQ as an underlying technology for message delivery among applications. Last but not least, there is a management console provided administration purposes.

Overall, it can be said that ActiveMQ is a well established mature project. It has a number of articles and sources which contribute to the support offered for the solution. In addition, it can be said that a number of configuration modes and possibilities exist for the system, which can be a benefit when deploying. However, it can also impose as a problem since there is a learning curve as to which functionalities are unnecessary. Having covered the essential aspects in relation to the work at hand, it deemed as a suitable solution when it comes to utilizing an open source messaging system.

6.2.1.2. RabbitMQ

RabbitMQ is an AMQP messaging broker developed and maintained by Rabbit Technologies Ltd. Rabbit is also a part of AMQP working group, thus, working on the future of message broker interoperability. There is a notable difference that RabbitMQ is implemented in Erlang, unlike ActiveMQ which is Java based. RabbitMQ supports a number of clients, three of which are considered as official, namely Java, .NET/C# and an Erlang client. However, due to the substantial community support there are a number of other clients for Python, Perl, Ruby and many more developed\textsuperscript{26}.

Similarly with the ActiveMQ, the two messaging models, of queues and topics, are well established within the solution, as well as the persistence mode and acknowledgements. It too is designed for providing clustering to achieve a single broker view. Unlike ActiveMQ, this solution has a flexible routing scheme which is as a consequence of adhering to the AMQP protocol. In the case for ActiveMQ this capability could be added by utilizing another Apache project named Camel. On the other hand, RabbitMQ is not governed by JMS, rather, its main governing point and goal is the adherence to the AMQP specification, however it does offer a STOMP plug-in as well. There is a management console for administration purposes, similarly to ActiveMQ. There are also a number of other plug-ins that are available that extend its functionality. RabbitMQ has similarly SSL support, however it does require using its own database for persistence purposes, rather than allowing the user to use its own\textsuperscript{27}. The number of protocols it supports is not like in ActiveMQ, due to the fact that the AMQP specification assumes a TCP connection.

It should be noted that the AMQP specification is still an ongoing process which means that RabbitMQ would have a number of changes in the future. However, regardless of this, the solution is established in the market as one that is trying to conform to the AMQP specification. Thus, it would mean that any future use would be beneficiary to the solution that uses it, such as the APF. These cases are important to note, due to value that they bring to

\textsuperscript{25}http://activemq.apache.org/features.html
\textsuperscript{26}http://www.rabbitmq.com/devtools.html
\textsuperscript{27}http://www.rabbitmq.com/features.html
both ESB and SOA. Since interoperability is a key to these architectures, standardized protocols, allow incorporation of other solutions that might already have existing MOM in place.

RabbitMQ was considered in tight relation to another Apache project named QPID, supported by the Apache Software Foundation. This project has two brokers, namely one in Java but there is also a C++ version of the broker. There are four available clients, which are Java JMS, C++, C# .NET, Python and Ruby, unlike RabbitMQ where due to its community support there are many more. Having already used one project by the Apache Software Foundation, the choice was made to take another technology in order to be able to differentiate them to a certain degree. The project is less mature than ActiveMQ, but RabbitMQ\(^{28,29}\) tries to adhere to the AMQP which is more important to consider in this case.

ActiveMQ is a mature technology that has been around for many years, having a wide community support which is especially important when it comes to open source solutions. It is highly configurable and a number of other projects exist offered by the Apache Software Foundation that could enhance its capabilities. The RabbitMQ messaging system on the other hand is aimed at adhering to the AMQP protocol rather than the JMS API, and has been established much later than ActiveMQ. It will succumb to changes in relation to the AMQP protocol specification in the future, however an important factor is the fact that it tries to adhere to it. Both are well established in the market, have excellent support, are reliable and aim at high performance.

For clarification purpose, table 4 serves as a summary of the comparison points and set of criteria that were identified in subsection 3.3.7.

<table>
<thead>
<tr>
<th>Features</th>
<th>MOM solution</th>
<th>ActiveMQ</th>
<th>RabbitMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available clients</td>
<td></td>
<td>Java; Erlang; JavaScript; Perl; PHP; Pike; Python; Ruby and Rails; Ajax; etc(^{30})</td>
<td>Java; Erlang; Perl; PHP; Pike; Python; Ruby ; Ajax; etc(^{31})</td>
</tr>
<tr>
<td>The size and activity of the Support community</td>
<td></td>
<td>Over 50,000 members in the Apache Community</td>
<td>Active community support</td>
</tr>
<tr>
<td>Communication protocols supported (such as TCP, UDP, etc.)</td>
<td></td>
<td>TCP, UDP, SSL, NIO, etc(^{32})</td>
<td>TCP, SSL(^{33})</td>
</tr>
<tr>
<td>Interoperability protocols supported (such as AMQP, OpenWire, STOMP)</td>
<td>AMQP not supported at the moment; STOMP; XMPP;</td>
<td>AMQP; STOMP extension; XMPP; (^{31})</td>
<td></td>
</tr>
</tbody>
</table>

\(^{29}\) [http://www.rabbitmq.com/changelog.html](http://www.rabbitmq.com/changelog.html)
<table>
<thead>
<tr>
<th>Feature</th>
<th>ActiveMQ</th>
<th>RabbitMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for SSL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Routing</td>
<td>With another project (Camel)</td>
<td>Yes, through exchanges (AMQP native)</td>
</tr>
<tr>
<td>Clustering</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4: Summary of Features for ActiveMQ and RabbitMQ

6.2.2. Implementation of Messaging Functionality

6.2.2.1. ActiveMQ

In relation the ActiveMQ solution, there was a possibility to use a JNDI global property file which would hold the necessary information for retrieving the Connection factory necessary for establishing the communication channel between the ActiveMQ broker and the clients i.e. both the Producer and Consumer of messages. Since ActiveMQ is based on the JMS specification the image below contains the programming concepts related to it. It should be noted that the legacy implementation of the APF also conforms to the Java JMS specification. This meant that a lot of the functionalities would be similar to the previous implementation, however there were changes as to the specific implementation API provided by ActiveMQ.

![Figure 22: JMS Programming model Overview](http://activemq.apache.org/protocols.html)

In the specific case of the ActiveMQ messaging solution API, the ActiveMQConnectionFactory class names are being retrieved from the properties stored in

---


35 Image taken from: (Curry, 2004)
the JNDI. This class is used to create the Connection object to the existing messaging broker or provider, as it can be referred as well. The Connection object will later on represent the Producers (client) connection to, in this case, the ActiveMQ message provider. After the connection has been established to the specified broker, the object is used to create a Session object. Through the Session object one can create both Producers and Consumers of messages for both sending and receiving respectively. Additionally, queues, topics and messages are created through this object as well.

The destinations (queues) are created through the use of the configuration files of the specific APF instance. Each of the four identified queues are declared in a configuration file and the information is used in order for them to be initialized with the specified names. After a lookup has been performed in the JNDI, the queue is created dynamically together with the declared names in the APF configuration files. The Session object, as it was previously mentioned, is used for the creation of the queues.

The Producer is created through the Session object as well, as it can be seen on Figure 22. Once the queues have been declared, the created Producer object is used in order to send the messages over the connection that was established. In addition, the persistence mode is set through the Producer object, that governs if messages are stored or not, which by default is to persist. This was considered as a reasonable choice due to the fact that it guarantees delivery, thus not losing any messages that are exchanged between the instances. Those are the steps required for the creation of a Producer, responsible for putting messages on one of the identified messaging channels within the APF. The consumer side (client) is done in a similar manner as it will be seen in the following steps.

The steps required are similar as with the previous case for the Producer, with the notable difference of creating the Consumer object instead of the Producer object. This means that the creation of the ConnectionFactory, Connection, Session and Destination (queues) objects remains the same. It should be noted here, that the configuration files within the other instance of the APF, i.e. where the Consumer would be, contain different queue names. This way a two way communication channel can be established between communicating APF instances. The Producer and Consumer objects need to be created and attached to different destinations (queues) for sending/receiving purposes. The other major difference is that the consumer object is used for listening for messages that have arrived on the specified message channel; in the case of the Producer object it was used to send messages.

6.2.2.2. RabbitMQ

In addition to the previous example the RabbitMQ solution needed to be integrated with the APF as well. As it was mentioned in the comparison subsection earlier (6.2.1) the RabbitMQ broker is not tied to the JMS specification, rather it is based on the AMQP protocol. The client that was utilized was in Java, similarly to the case with ActiveMQ. Essentially there were not that many dissimilar concepts when it came to the implementation. One of the initial differences between the two solutions is the fact that the RabbitMQ does not provide a JNDI service. This was circumvented by configuring one of the files that are utilized by the APF to specify where the broker is located.
Proceeding with other concepts within the RabbitMQ solution, the ConnectionFactory concept remains as well as the Connection object. The Connection object is used for the creation of the Channel object, over which the communication takes place with the RabbitMQ broker. The Channel object can be related to and is similar to the Session object from ActiveMQ. On the specific Channel object, a queue needs to be declared, again, by utilizing the configuration files from the APF. In the case of RabbitMQ there is no explicit Producer object. Before proceeding with the next step of publishing the message on the channel, it is necessary to specify the properties for persistence through a BasicProperties object. As it was the case in ActiveMQ, the messages are persisted here as well, so they would not be lost. Afterwards, the publishing of the message goes through the Channel object itself, rather than a Producer object, like in the case of ActiveMQ.

In relation to the Consumer side of the story, there are also some notable differences. The ConnectionFactory, Connection and Channel objects are initialized similarly to the previous case for publishing the messages. Now for receiving the messages, a QueuingConsumer object needs to be created that blocks until the next delivery by the RabbitMQ broker. The QueuingConsumer is created with a specific channel, and then through the Channel object a Queue and a Consumer are tied together. This can be seen as from the code snippet below.

```java
QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(QUEUE_NAME, true, consumer);
```

The QueuingConsumer has a Delivery object that contains the actual message received from the broker. There is a notable difference between the two solutions when it comes to message types. Within ActiveMQ there are several different message types identified, such as TextMessage, BLOB message etc. In the case of RabbitMQ information is sent only as bytes message. The Delivery object contains the bytes which then need to be read to the necessary object.

This concludes the implementation detail in regards to the two selected open source MOMs. As it could be seen, there are certain common points among the two solutions. However, there are also other that are substantially different, such as the protocol used, the different object and classes. The chapter continues with the implementation details regarding the applications.

### 6.3. Application Implementation and Integration

In order to provide an example of a simple integration, using the APF, two applications were implemented which communicate over this framework. Subsection 6.3.1 discusses the idea behind the applications and the process of implementation, while subsection 6.3.2 presents the integration process.
6.3.1. Implementation

The applications were created based on the concept of Requester and Provider, meaning, the first application sends a request to the Provider, which is followed by returning an appropriate reply. In that manner, the logic behind this interaction was for the Requester to ask for RSS feeds from a specific author, while the Provider to check his list of feeds and returns all which were written by the requested author, in an RSS format. The Requester has also implemented an RSS reader, so it presents the RSS feed received in the reply as an XML, into well structured, presentable format. The functionality of the applications is very simple, however, sufficient for presenting the functionality of the APF in practice. The main point through this example is showing how two applications can communicate, although both have their own application specific message format.

Basically, when the Requester sends a request, it provides the name of the author in the following format:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<author>
    <Name>James</Name>
    <Last_Name1>Collin</Last_Name1>
    <Last_Name2>Brown</Last_Name2>
</author>
```

**Code Snippet 3: Request format on the Requester’s side**

However, let’s say that the Provider has a “legacy” code and was implemented to receive the requests in a different format, such as:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<author>James Collin Brown</author>
```

**Code Snippet 4: Request format on the Provider’s side**

The same occurs on the way back, when the reply is sent from the Provider. Once again, the Provider sends the collection of RSS feeds from the requested author, having the author tag, within, in the same format as the requests it expects:

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="2.0">
    <channel>
        <item>
            <title></title>
            <link></link>
            <author>James Collin Brown</author>
            <description></description>
        </item>
    </channel>
</rss>
```

**Code Snippet 5: Response format on the Provider’s side**
However, this format is not understandable by the Requester. What he is able to process further is his own representation of the author:

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="2.0">
  <channel>
    <item>
      <title></title>
      <link></link>
      <author>
        <Name>James</Name>
        <Last_Name1>Collin</Last_Name1>
        <Last_Name2>Brown</Last_Name2>
      </author>
      <description></description>
    </item>
  </channel>
</rss>
```

**Code Snippet 6: Response format on the Requester’s side**

As it is visible, the provider and requester do not understand each other’s formats, so if one receives the information in the wrong format, an error will occur. This suggests that these two applications are not compatible for communication. However, the APF comes here in place. It not only establishes a communication channel between the applications, but converts the message formats into the ones understandable by the applications.

Having presented the main idea behind the implemented applications, their purpose, as well as the data format they operate with, it can be proceeded to elaborating the details behind the integration between these two applications.

**6.3.2. Integration**

When it comes to integrating applications with the APF, the first step is to actually connect the application to an instance of the framework. As it is implemented, each application has its own instance of the APF, which can communicate with other APF instances. By APF instance it is meant a group of the three main APF components. For the purpose of connecting an application to the framework, a dedicated connector is implemented. It connects the application with the AIC component, by exposing the functionality of the AIC to the application. Thus, when the application creates an instance of this connector, it is able to send a message to the AIC component for further processing.

After having the connector in place, the application starts by preparing a request message, which, at the time, is a Java object. Then it uses the connector to call a method of the AIC and pass the request message to it. When the message arrives at the AIC, in this component, it is converted from Java object into XML format and passed to the TVE component.
As it was discussed in the literature review, more specifically in the section on normative data, the most common and suitable way of handling different formats is establishing a Canonical Message Model (CMM). Thus, the message sent by the requester will therefore be transformed from the requester specific message format into the CMM, before sending it off to the provider. It can be argued that, when only two applications are integrated, a CMM is not necessary and it would create unnecessary overhead; however, in this case, it was introduced for the purpose of presenting the bigger picture, and what would be the process, if more than two applications are being integrated.

Therefore, the transformations that are supposed to be executed on the request message are listed in the TVE configuration file, under the list of actions. It is important to mention that different transformations are needed for the request and the reply; however, there is only one configuration file from which they are read. For that purpose, a flag was introduced noting whether the action of the TVE is for an outgoing message (which in the case of the Requester is a request) or incoming message (a reply for the case of the Requester).

In the specific case implemented, on the request message, a transformation of the message format is applied. As it was seen from Code Snippet 3, that is the requester specific format. On the following code snippet, the CMM format is presented, which the request needs to be turn into:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<author>
  <Name>James</Name>
  <Last_Name>Collin Brown</Last_Name>
</author>
```

**Code Snippet 7: CMM format**

For that purpose, a transformation module of the TVE component was used, which is responsible to transform an XML (in this case the request message), using XSLT, in a different format (in this case the CMM). The XSLT developed for this purpose can be seen bellow:

```xml
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="author">
    <author>
      <Name>
        <xsl:value-of select="string(Name)"/>
      </Name>
      <Last_Name>
        <xsl:value-of select="string(Last_Name1)"/>
        <xsl:text>
          <xsl:value-of select="string(Last_Name2)"/>
        </xsl:text>
      </Last_Name>
    </author>
  </xsl:template>
</xsl:stylesheet>
```

**Code Snippet 8: XSLT for transforming Requester’s specific to CMM formatted request**
In that manner, when this XSLT is applied to the request message, the CMM format is the result (Code Snippet 7).

When the transformation is completed, the message, now in the CMM format, is passed on to the MPAS component. As it is known from Chapter 4, the MPAS component is responsible for dispatching the message off to another APF instance, and also for listening on the IN queue for incoming replies. Thus, when the message arrives to the MPAS, before actually sending it, a header is appended on the message. Currently, the header keeps some basic information regarding the message, such as the message type (request, reply, error, etc.) for example. However, in the future, the header will also keep routing information, when the routing functionality will be implemented in the APF.

After the header is attached, the request message is put on the OUT queue and sent off to the Provider. In continuation, the same request is also saved within the Message Warehouse, discussed, as well, in Chapter 4. Finally, the execution reaches the point where it blocks and waits for a reply. This part concludes the sending of the request.

At the same time, the Provider, on the other side, was waiting for an incoming request. Thus, when the sent message reaches the Provider, it first enters the MPAS component. Initially, the header is extracted from the message, but the messages itself is once again stored in the message warehouse. Continuing, the received request is passed from the MPAS back to the TVE component. At this point, the message has the CMM format, however, in order for the provider to be able to work with it, it has to be transformed to the provider’s specific format, which was presented in Code Snippet 4.

In the implemented example, on the Provider’s side, the TVE has two actions which are to be executed on the request. One is transforming it into the Provider’s format, and the other is validating the output of the transformation. Starting from the transformation, again, the transformation module of the TVE was used for this purpose, which converted the CMM format presented in Code Snippet 7 into the provider’s specific one. For this purpose the following XSLT file was used:

```xml
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="author">
    <author>
      <Name>
        <xsl:value-of select="string(Name)"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="string(Last_Name)"/>
      </Name>
    </author>
  </xsl:template>
</xsl:stylesheet>
```

**Code Snippet 9: XSLT for transforming from CMM to Provider’s formatted request**
After completing the transformation, the transformed message is validated within the validation module of TVE, which checks if the format of the message is according to the provider’s format. The XSD schema used for this purpose is the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="author" type="xs:string"/>
</xs:schema>
```

**Code Snippet 10: XSD for validating the transformed request on the Provider’s side**

As mentioned before, this schema checks if the transformed message has only one element – author, and within has a string, which would be the author’s name. When the validation is successful, and thus, all the actions within the TVE component are completed, the message is passed to the AIC component.

This time, since the message is in XML format, it needs to be turned back into Java object. After this conversion is completed within the AIC component, through the connector with the application, the readymade message into Java object is passed to the Provider. Here in the actual application, the Provider reads the request embedded in the object, does the necessary processing, in terms of finding all the RSS feeds from the author sent in the request, and prepares the reply.

Next, the reply is sent through the connector to the AIC component, where, from Java object is converted to XML message. At that point, the reply has the following format (which, again, is the provider specific):

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="2.0">
  <channel>
    <title>My Feeds</title>
    <link>http://www.jcb.com</link>
    <author>James Collin Brown</author>
    <description>I am starting my rss feed service today</description>
    <item>
      <title>Day 1</title>
      <link>http://www.jcb.com/day1</link>
      <author>James Collin Brown</author>
      <description>RSS for dummies</description>
    </item>
    <item>
      <title>Day 2</title>
      <link>http://www.jcb.com/day2</link>
      <author>James Collin Brown</author>
      <description>Already an expert?</description>
    </item>
  </channel>
</rss>
```

**Code Snippet 11: Reply in Provider’s format**

When the conversion in the AIC is completed, the reply is sent to the TVE component. Here the transformation from the provider specific format to the CMM, takes place. For this purpose, the XSLT file in Appendix A, Code Snippet 12, was used, and the output of the
transformation can be also found in Appendix A, Code Snippet 13. This transformed reply, is then sent to the MPAS component, where the header is attached, and it is sent off to the Requester.

On the requester’s side, the reply is received again by the MPAS component, where the header is extracted, and the reply is passed to the TVE. Here, the message is transformed from the CMM format, to the Requester’s specific format. Again, the XSLT file for this purpose, as well as the result of the transformation can be found in Appendix A Code Snippets 14, 15 respectively.

After the transformation, there is also a validation action that needs to take place in the TVE. This was added for the purpose of checking whether the transformed reply message follows the requester’s format. The XSD file used within the Validation module can be found in Appendix A, Code Snippet 16.

Eventually, when the message passes the validation check, it is sent over to the AIC component. Now, the opposite conversion occurs here – the message is transformed from XML format to Java object, which is finally returned, through the connector, to the Requester application. Here the Requester passes the object to its RSS reader, which prints the details of the RSS file received as a reply, nicely on the screen.

Overall, this subsection covered the details behind one complete request-reply cycle of interaction between the two integrated applications. The visual representation of the message flow can be also found on Figure 23.

![Figure 23: Message Flow](image-url)
7. Evaluation

The following chapter summarizes the steps and the processes undertaken to complete the evaluation for both the message queuing solutions and the APF. In order to successfully accomplish the objective, it was divided in two steps. First, the throughput of each of the used message queuing solutions was tested. Second, was the overall evaluation of the component architecture i.e. the APF itself, without taking into consideration the execution time of the message queuing solution. There are two main reasons as to why the task was divided in such a manner as it will be elaborated in the following paragraphs.

The reason behind the first step was to evaluate the “raw” throughput of both message queuing solutions in order to be able to perform an evaluation of the two. For this purpose two separate benchmarking programs were created that would enable automating the process of evaluation. This enables testing each individual message queuing solution in a number of arrangements and with greater precision, in order to add value to the evaluation results. In addition, the task was further divided in two more steps by evaluating the performance on a single machine and on two separate machines. For the purposes of the first evaluation scenario, the broker was running together with a single producer and a single consumer on the same machine. The second case, involved dividing the broker and the consumer on one machine and the producer on another machine.

Besides benchmarking the message queuing solutions, the second step involved also benchmarking the APF. This involved adjusting the APF solution in order to add a number of measuring points or time flags that would enable the performance evaluation of the solution. This would include both testing the overall performance of the APF but also the performance of the individual components. All test results were saved in a CSV file for further processing of the information. The time flags were added within the APF solution and are not covering the time it took for the exchange of messages over the message queuing solutions.

This is where the benefits of separating the evaluations tests in two steps can be seen. One benefit was that the message queuing solutions could be compared in relation to each other separately. This would extend the comparison between the two message queuing solutions that was presented in subsection 6.2.1. The results can be used to see which of the two could be a more suitable option as an underlying technology. The other benefit is that the APF could be evaluated, regardless of the message queuing solution utilized. This would enhance the knowledge and insight into the framework itself and serve as a base for further improvements of the framework.

In relation to the steps that were mentioned, the chapter was organized in a similar manner. The section that follows, elaborates the reasoning behind the division of the work, as well as the description of the benchmarking programs for both steps. The chapter continues with a section regarding the testing process that took place. Last but not least, the last section summarizes the results from the tests that were performed. This again, entails both results for the message queuing solutions and the APF also.
7.1. Description of the benchmarking programs

The following subsections explain how the performance was calculated for the messaging solutions as well as the details regarding the APF evaluation.

7.1.1. Messaging solutions performance evaluation

After covering the general description of the steps that were taken in order to complete the evaluation, the following subsection will try to elaborate the approach taken for the programs. In the first step, the benchmarking programs follow a pattern of measuring each solution at particular points. The tests were done for both ActiveMQ and RabbitMQ and the messaging solutions were used as is, i.e. with the default configurations and no additional changes to the configurations were made for improving the performance.

There is a single producer and consumer in each of the scenarios sending/receiving messages over a single queue. The messages are auto acknowledged upon receipt by the consumer and the size of the messages is always constant at 13 Bytes. There is awareness that, in actual production environment the message could be variable and larger. As discussed with Aurenav LLC, the message size, when the solution was deployed in the banking and financial sectors, there was a noticeable difference, starting from several K to several hundred K messages. What varies is the number of messages that need to be exchanged, the number of computers used and if persistence mode is enabled. In regards to the performance of the messaging solution, three main variables were considered:

- The throughput which the producer/consumer achieved, measured in messages per second.
- The total time it took from sending the first message until the last message was received.
- Last, at the consumer side, the average latency for the delivery of the messages was measured.

The generic formula used for calculating the throughput was dividing the number of messages that were sent by the time it took for their transfer. The throughput was measured at two different points: one point was at each second to follow the overall performance during execution time and one at the end of the performance test for reporting purposes. For the first scenario, whenever a one second trigger was started, it was calculated how many messages have passed within a time interval. Thus there are the following four variables:

- currentNumOfMsgs – an increasing counter representing the latest number of messages sent/received at the time of reporting
- previousNumOfMsgs – an increasing counter representing the number of messages sent/received at the previous reporting time
- currentReportTime - time marked at the moment of reporting
- previousReportTime - time of the last report
In addition, the throughput can be calculated as seen on the formula below:

\[
\text{Throughput} = \frac{\text{currentNumOfMsgs} - \text{previousNumOfMsgs}}{\text{currentReportTime} - \text{previousReportTime}}
\]

In addition, at the end of the test, the final throughput is calculated in a similar manner, with the notable difference that the total number of messages was used over the total time it took from the start to the end of the test. Last but not least, the latencies were saved for each of the messages so that they would be compared. The latency for each message was calculated by having the difference between the time marked at sending of the message and the time marked at receiving it at the consumer side. This helped in evaluating the average latency which was calculated in the following manner. First of all if there is:

- \( t_{Send} \) - Time marked at sending a message
- \( t_{Recv} \) - Time marked at receiving a message

Then the latency can be calculated in the following manner:

\[
\text{Average Latency} = \frac{\sum_{i=1}^{150000} (t_{Recv_i} - t_{Send_i})}{n}
\]

It should be noted that all the data processing was performed after the execution time of the tests so it does not influence the performance.

### 7.1.2. APF performance evaluation

Besides the benchmarking tests of the message solutions, the APF was adjusted to facilitate the evaluation process as well. For this purpose, there were four identified measuring points within the framework for both the requester and the provider. As it was previously mentioned in section 6.3.:

- **Requester** is the application that generates a request and sends it off to another APF instance, from which later on it receives a reply;
- **Provider** is the application that waits for request and based on the received request, generates a reply that sends it off to the application, from which it received the request in the first place;

Within the requester, the first point is noted at the beginning, when the request is received in the AIC component. Second point is when the request has reached the final MPAS component before putting the message on the channel. As it was mentioned, the actual delivery and exchange of the messages was not measured at this step. Rather, when a reply message was received the third point marked the time of the message arrival within the MPAS component. Last but not least, the fourth point was when the reply reaches the AIC component, before delivering it to the application.

Similarly, it was done for the provider, by having again four measuring points. The first point is at the receipt of the request by the producer. The second point is at the AIC
component level to measure the time taken for the request to be processed and delivered to
the application. There is another division here to avoid measuring the processing time
necessary for preparing the reply. The third point is from the AIC component when the reply
message is being sent through the components. The last point is in the MPAS component,
before putting the reply on the channels.

In addition, to the tests described previously, the evaluation process went into more
details of testing the actual performance of the individual components that comprise the APF.
For this matter, similarly to the previous description, when a message entered a particular
component, the time was marked, as well as when the message left the component. The
measurement points were set for each of the components, namely AIC, TVE and MPAS.
These measurement points would mark the start and completion time for the message to pass
through the component giving valuable insight into the performance of the individual
components.

7.2. The Test Process

The last step after describing the reasoning and the details of the benchmarking programs,
is to elaborate the tests that were actually performed. All tests were performed with both
message queuing solutions that were discussed, namely RabbitMQ and ActiveMQ. It should
be noted from the beginning that the APF was, previously, implemented using commercial
software. Typical usage was in banking and financial services environments,
with high number of transactions that were successfully handled by the APF (Baldwin, 2012).
In regards, to the testing environment that was used internally for the open source solutions,
it consisted of the following two Notebook machines specifications:

**Dell Latitude E5420**

- Processor: Intel(R) Core(TM) i3-2310M CPU @2.10GHz
- RAM: 8GB
- 64-bit OS Windows 7 Professional

**HP Pavilion dv6**

- Processor: Intel(R) Core(TM) i5-450M CPU @2.40GHz
- RAM: 4GB
- 64-bit OS Windows 7 Home Premium

Any additional software that was not of particular use by the APF, was stopped due to the
effect that it might have on the performance. Moreover, the previous deployment scenarios of
the APF consisted of multiple HP servers with dual or quad-core CPUs. In addition, the
environment used containers and clusters with multiple APF component instances to achieve
greater scalability (Baldwin, 2012).
7.2.1. Message Queues

For the purposes of evaluating the message queuing solutions a number of tests were performed. There are three variables that were modified to perform the tests. These are the number of messages that were sent/received, persistence mode of the messages and the number of machines. The tests were performed with 50,000, 100,000 and 150,000 messages and for each of these test cases the tests were done with both persistent and non-persistent mode. In total, there were 24 test cases performed on the Dell machine, counting for both the producer and consumer, for both message queues.

In addition, it should be noted that the test cases were done in order to be able to evaluate and compare which test set would suit the needs of the evaluation process. After the initial tests were performed, the 150,000 messages test set was chosen since it was the most informative in terms of results to be presented and due to practicality reasons that not all performance tests could be presented in this document. In addition, these test cases were running the longest (more than 15 seconds) in relation to the others, which adds more value to the results, rather than having test cases that run shorter.

The last part of testing the messaging solutions was to execute the benchmarking programs on two separate machines. This was done by executing the message broker and consumer on the Dell machine, while having the producer on the HP machine sending messages, communicating over 1Gbit Ethernet. For the purposes of these test cases using both machines, the 150,000 testing set was used again in both persistence and non-persistent mode, as it was most suitable. In total there were 8 test cases for this part, again counting for both producer and consumer. Dividing the test cases in such a manner enabled the evaluation of the message queuing solution, regardless of the performance of the APF components, which helps in comparing them, but, also, for future considerations on which is more suitable for the APF.

7.2.2. Access Point Framework

The testing case was modified as well, when it came to evaluating the APF. The number of messages was set to 3000 for the overall performance test and 1000 for testing the individual components of the APF. The time noted represents the duration of the actual message flow between the components rather than the time it took for message exchange over the messaging channels. After performing the APF tests, the data was utilized to get meaningful results, which are presented in section 7.3.2. that follows later on. The tests were carried out on two machines with no previous tests done on a single machine, due to the architecture itself and the intention to use the APF in a distributed environment. The requester was running on one machine whereas the provider was situated on a different machine.
7.3. Results

This section discusses the results obtained from the tests conveyed and described in the previous sections.

7.3.1. Message Queues

This subsection discusses the results obtained from the testing of the message queues. They are logically separated on results from tests on single machine and two machines. Eventually a comparison between these results is provided as well.

7.3.1.1. Results from single machine

The results presented in this subsubsection were obtained by running the programs on a single machine. Producer is the program that generates the messages and sends them on the queue, which can be either persistent or non-persistent. Based on this, the producers and consumers are called persistent or non-persistent. Additionally, the consumer is the program that receives the messages on the other end of the queue.

![Throughput](https://via.placeholder.com/150)

**Figure 24: Throughput measured for ActiveMQ and RabbitMQ**

On Figure 24 the throughput, or the number of messages per second produced and consumed, of ActiveMQ and RabbitMQ is presented. As it can be seen, the non-persistent producer of ActiveMQ has almost twice as bigger number of messages produced per second when compared to Rabbit MQ. The case is very similar for the non-persistent consumer, too. However, when it comes to Persistent Producers and Consumers, the number of messages sent and consumed by ActiveMQ drops drastically. For RabbitMQ the number of messages is insignificantly smaller when compared to the numbers of the non-persistent producer and consumer. Therefore it can be concluded that regarding the throughput, there is great difference for ActiveMQ in the performance when it comes to persistent queues, while RabbitMQ has very similar performance despite the persistence.
Figure 25: Average Latency measured for ActiveMQ and RabbitMQ

On Figure 25 the average latency of the two message queues is presented, both in persistent and non-persistent mode. As it can be noticed, RabbitMQ has similar and low latency in both modes, while there is drastic difference in the case of ActiveMQ. For persistent queues, ActiveMQ has lower latency than RabbitMQ, however, in non-persistent mode, the latency of ActiveMQ is extremely high, compared to all other results.

From this graph, together with the results from the throughput, it can be concluded that in non-persistent mode, ActiveMQ has very high throughput, but also high latency. For persistent queues, the throughput is noticeably lowered but the latency is improved. RabbitMQ, on the other hand, keeps the balance steady – there is no significant difference of performance in terms of throughput and latency in both modes.

Figure 26: Execution Time measured for ActiveMQ and RabbitMQ
The execution time necessary for sending 150,000 messages was measured as well. The results are presented in Figure 26, in seconds. Proportionally to the results of the throughput, there is almost no difference in the times measured for Rabbit MQ to complete the transfer of the messages. Both consumer and producer need around 39 seconds to complete the sending/receiving of messages, regardless whether the queue is persistent or not.

Nevertheless, in the case of ActiveMQ, the time for completion in non-persistent mode is much faster than RabbitMQ, since it manages to transfer the messages in about 23 seconds. However, for the persistent queue, it took 525 seconds for the same amount of messages, which was expected, considering that the throughput in this mode was very low as well.

![Non-Persistent Consumer for 150K messages](image)

**Figure 27: Non-Persistent mode for ActiveMQ and RabbitMQ**

In order to get a closer overview of the performance of the non-persistent consumers, the graph on Figure 27, was created. The purpose for showing only the consumer is due to the fact that the results with the producer were almost identical. Now, as it is visible on the graph, Active MQ manages to send more messages per second and completes the transfer faster than RabbitMQ, however the throughput varies visibly through time. On the other hand, RabbitMQ takes longer and has lower throughput than ActiveMQ, but keeps it steady throughout time.
As expected, the results in persistent mode are radically different. On Figure 28, it can be noticed that RabbitMQ has both higher throughput and manages to transfer the 150,000 messages in much shorter time. ActiveMQ has much worse throughput (which varies again), as well as execution time.

### 7.3.1.2. Results from two machines

Besides conveying the tests on one machine, the same were executed in real distributed environment, meaning on two different computers. The results obtained are discussed further in this subsubsection.

The overall results from Figure 29 were visibly improved for both message queues. Again, ActiveMQ outperformed RabbitMQ in non-persistent mode, while RabbitMQ showed much better performance than ActiveMQ in persistent mode. Similarly, RabbitMQ keeps the same throughput rate in both persistent and non-persistent mode, while ActiveMQ has drastic difference.
Figure 30: Execution time for producer and consumer on separate machines

Due to the improved performance when running on separate machines, the execution time necessary for transferring 150,000 messages, was lowered as well, as it can be seen from Figure 30. In non-persistent mode the results are quite similar for both queues, having ActiveMQ as the one that took less time due to the higher throughput, while in persistent mode, RabbitMQ completed the transfer much faster.

Figure 31: Execution flow for non-persistent consumer
Figure 32: Execution flow for persistent consumer

On Figure 31 and 32, the throughput over time can be seen in both non-persistent and persistent mode for both queues, correspondently. On Figure 31, the higher throughput and faster completion of ActiveMQ over RabbitMQ can be seen, while in persistent mode, Figure 32, it is the opposite case.

7.3.1.3. Comparison one vs. two machines

Having presented the results of execution on one machine and on two machines, it can be useful to compare them and see the actual difference.

Figure 33: Persistent consumer on ActiveMQ-comparison on local and two machines
On Figure 33, the performance of ActiveMQ in persistent mode, having the producer and consumer on single and on two machines, is presented. As it is clearly visible, both the throughput and execution time needed, show better results, having the programs running on two machines. The case is similar with the persistent mode of RabbitMQ (Figure 34). Running the producer and consumer on separate machines showed better results than having them on a single computer.

Eventually, for the non-persistent mode, the results of both queues are presented on a single graph in Figure 35. Of course, the best performance measured was achieved by ActiveMQ, having the producer and consumer running on separate machines. The second best is RabbitMQ, also with the programs deployed on two computers. Further comes ActiveMQ on local machine and last is RabbitMQ on a single machine as well. From the graph is, of course, visible the throughput over time for each deployment model, as well as the execution time needed to complete the transfer of 150,000 messages.
7.3.1.4. Discussion on the Message Queues Performance

So overall, having the results presented above, it can be seen that having the programs deployed on separate computers, increases the performance of the queues. The reason for this is that when they are deployed on a single machine, both consumer and producer consume great amount of the resources of the machine, and so as the broker running on it. Therefore, in order for the programs to run, they have to share and balance the resources of the machine, which means, that they will have to lower their performance in order to coexist. In that manner, when one program (the consumer) was moved to another machine, more resources were available to all three (including the broker). Respectively, this resulted in improved performance and better results when running on separate machines.

Accordingly, this led to a very important realization, which can be considered as a limitation to the tests performed. The results presented in this chapter do not realistically reflect the capacity of the brokers of the two evaluated message queues. Rather, the results heavily depend on the capacity of the computers where the programs are running. In that manner, the results presented here reflect better the difference in performance between the two message queues, rather than their actual capacity.

Having gathered and discussed all the results, if the performance is taken mainly into consideration, it can be safely concluded that for solution in need of non-persistent messages, ActiveMQ will bring the best results, while for persistent messages, RabbitMQ is definitely a better way to go with. For solutions that may require both persistent and non-persistent messages, again RabbitMQ would be more suitable due to its balanced and similar performance in both modes.

When choosing messaging solution, performance might be one of the most important aspects and for that purpose a lot of attention was dedicated to it. However, it is not the only aspect that should be taken in consideration, and for that purpose, more details on these two message queues can be found in Chapter 6 subsection 6.2.1.

7.3.2. Access Point Framework Performance Results

As described in subsection 7.2.1, the performance of the APF has been evaluated as well. On Table 5 and Table 6, part of the obtained results are presented. In order to better understand the representation, it is important to mention that:

- **Send Request** signifies the average measured time for the requester to pass the prepared request through the 3 APF components, prior to sending it off to another APF instance
- **Receive Reply** stands for the average time measured from the point when a reply is received, then passed through the 3 APF components until it is dispatched to the application
- **Receive Request**, similarly, is the average time for the received request to pass through the 3 APF components
- **Send Reply** is the average time needed to pass the prepared reply, through the APF before sending it back to the requester.

The results on Table 5 were obtained from the test conveyed with a set of 1000 messages, and they represent how much time, in milliseconds, is necessary for a message (request or reply) to pass through each of the three components of the APF. In the specific case tested, the times presented in the AIC column describe how much time was needed for the request/reply to be transformed from Java to XML and vice versa, since this is currently the functionality of the AIC. Furthermore, the milliseconds measured for the TVE component, describe the time required for two actions, when it comes to sending a request or reply:

- a Velocity transformation
- an XML transformation from application specific to the Common Messaging Model (CMM)

On the other hand, the times presented in receiving a request or reply for the TVE, are measured against other two actions:

- transformation from the CMM to the application specific format
- an XSD validation against a certain schema, for receiving a request or reply.

Lastly, the values presented in the MPAS column, describe the time necessary for adding a header to the sending message, placing it on the queue, and storing it in the message warehouse (for sending request or reply). Accordingly, for receiving request or reply, the times measured represent removing the message header from the received message and adding the message to the warehouse as well.

<table>
<thead>
<tr>
<th></th>
<th>AIC</th>
<th>TVE</th>
<th>MPAS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Send Request</strong></td>
<td>24.925</td>
<td>43.338</td>
<td>49.305</td>
</tr>
<tr>
<td><strong>Receive Reply</strong></td>
<td>32.363</td>
<td>30.774</td>
<td>19.021</td>
</tr>
<tr>
<td><strong>Receive Request</strong></td>
<td>63.957</td>
<td>28.154</td>
<td>16.878</td>
</tr>
<tr>
<td><strong>Send Reply</strong></td>
<td>20.212</td>
<td>34.203</td>
<td>26.139</td>
</tr>
</tbody>
</table>

Table 5: Time in milliseconds spent per component of the APF

In the lead, on Table 6, the results presented were obtained from the test with a message set of 3000 messages, explained in subsection 7.2.2. The numbers specify the total times in milliseconds necessary for sending a request, as well as receiving a reply, on the Requester side, and receiving a request and sending a reply on the Provider side. The times measured include all the actions previously described for each of the three components.
Table 6: Time in milliseconds spent per request/reply

<table>
<thead>
<tr>
<th>Average:</th>
<th>Requester</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Request</td>
<td>112.6867ms</td>
<td>/</td>
</tr>
<tr>
<td>Receive Reply</td>
<td>83.1077ms</td>
<td>/</td>
</tr>
<tr>
<td>Receive Request</td>
<td>/</td>
<td>112.7047ms</td>
</tr>
<tr>
<td>Send Reply</td>
<td>/</td>
<td>85.8457ms</td>
</tr>
</tbody>
</table>

On Figure 36, on the other hand, is presented the average time in milliseconds necessary for an APF instance to:

- send a request and process a reply; this is represented on the graph as **Requester**;
- process a request and send a reply; this is represented on the graph as **Provider**;

![APF Performance Measurements](image)

**Figure 36: Performance results for the Requester and Provider**

As it is visible, the results obtained are very similar, and the slight difference is due to the fact that the testing was performed on two different machines with different capabilities. So overall, it can be said that on average, for the **Requester**, 195.7943ms are necessary to perform all actions, including passing a message through the AIC, TVE, MPAS components in two directions. Similarly, for the **Provider**, on average it takes 198.5503ms to also pass the request and reply through the 3 components in both directions.

7.3.2.1. Discussion on the APF’s Performance

If the results obtained for the performance of the APF are taken into consideration, an average number of interactions between the requester and provider, _per second_ can be calculated. Therefore, the sum of the two final measurements (195.7943 for the Requester and 198.5503 for the Provider), would be 394.3446ms for sending and receiving request/reply. That means that approximately _two_ complete cycles of _sending request_ → _receiving request_ → _sending reply_ → _receiving reply_, running on two parallel APF instances,
can be completed within a second. This measurement was calculated having in mind that the current implementation of the APF is single threaded and synchronized, meaning that one cycle has to be completed before another one starts.

The final result of 2 request-response interactions per second might not be very high for an Enterprise Application Integration Framework. However, some aspects have to be taken into consideration. One is that the two APF instances were tested on PCs, where in reality for these purposes usually servers will be used, which offer bigger processing power. In a production environment deployed by Aurenav LLC, the APF instance handled throughput of several thousand messages, of variable size, per second, successfully, as the environment was different (Baldwin, 2012).

Another point is that from the deployment models of the APF discussed in Chapter 4 section 4.2, the one currently implemented and tested is the Centralized Integration model, which is suitable for small number of applications with low bandwidth and processing overhead. Therefore, for small business, with not so many applications to integrate, as well as not so many transactions per second, this can be quite appropriate solution.

Of course, we expect that the results will drastically improve when the APF is deployed in distributed manner. The results obtained in the evaluation of the message queues are real proof of improved performance, when running in distributed environment, where the throughput visibly and radically increased.
8. Conclusion

The main goal of this project was to present how the APF offers solutions to the potential weaknesses of the existing ESB products on the market. This was presented throughout the description of the AFP, its architecture and components, and mainly through the comparison with two representatives of the ESB market- Mule ESB and BizTalk ESB Toolkit. In both cases, the APF has more scalable design, with very flexible and simple architecture, which could overcome the problems discussed in the problem definition.

The second goal of this project was to integrate into the APF two open source message oriented middleware products, which in this case were ActiveMQ and RabbitMQ, and provide evaluation of their features, for the potential users of the APF to be able to choose the one that will fit their needs better. As part of the goal achievement process, two simple applications were integrated with the APF, for example purposes.

Finally, performance evaluation was done on both message queues and the APF itself. The performance testing of the message queues was done with the purpose of facilitating potential users in their choice of the MOM. On the other hand, the performance results of the APF were meant for showing what can be expected with the current implementation of the framework, meaning in the centralized, synchronous and single threaded version. The final number of two complete interaction cycles between the requester and provider, per second, are not very high at this stage, but the results are expected to increase when the framework will be fully implemented with the possibility of the distributed deployment model.

Eventually it can be said that the idea behind the APF, as well as its design, are worthwhile for further implementation, in order to fully exploit all of their benefits. The current implementation is merely a proof of concept rather than a final product, however, it is a good basis for building up the whole framework. Some of the ideas for improvement are discussed in the future work, and if sufficient effort is invested in their development, this framework will be able to stand shoulder next to shoulder with the leading products on the ESB market.
9. Future Work

As it was discussed throughout the paper, the APF is conceptually a great design, however, it still needs extensive implementation in order to be usable for other types of scenarios that would require enormous throughput rates. One of the first things that needs to be followed through and that has been planned for future work is connecting the components with message queues and adapting the code to support this functionality. Having this step completed will allow for actual distribution of the components on separate machines.

Of the same importance would be to add support for routing. This would be closely tied to the messaging solutions in place, and the background research has already been made in order to see what would be necessary in order to introduce the routing functionality. For ActiveMQ, another Apache product would facilitate this process, which is Apache Camel. This open source solution offers Java based API for defining routing rules, besides the rest of its functionality. RabbitMQ, on the other hand, does not need additional support from other product. It already offers the routing functionality, which can be incorporated simply through the code. This would encompass the initial step when it comes to routing, however, it would be beneficial to afterwards extend the functionality of the routing with configuration files. This way, the integration of new routing rules would be eased which would lead to faster integration with new applications. In addition, it would correspond to the existing configurable architecture of the APF components. So overall, it can be said that routing is one of the initial steps in the future work and it should be fairly easy to introduce.

Having a distributed environment would impose new problems and limitations that need to be considered and addressed properly. One aspect that could be enhanced is the security environment surrounding the APF. There are three major considerations when it comes to security. First of all, the framework could be extended with the use of SSL protocol for the messaging channels. The second idea for improving the security of the framework is adding another specific functionality to the security module within the TVE component. An extensive work within Aurenav LLC. has been done with AuthAnvil Two Factor Authentication by Scorpion Software, which is software for user authentication using tokens. In that manner, it is one of the future plans to integrate AuthAnvil2FA into the APF as one of the security modules. The third step would involve encrypting the information that is being sent over the message channels. All these steps need to be optional, based on the security demands of the end users, meaning that the configuration files need to be modified.

A very important issue that needs to be addressed in the future work is the support of different message types. As it was discussed, the APF currently supports only TextMessages, however, it is important to be able to send Files or Objects throughout the framework. One of the biggest benefits of this framework is the fact that, in case of large files, it can offer easy setup of point to point connection, circumventing the broker. Thus, in order to exploit fully this advantage, adapting the framework to support file transfer is essential.
Due to the complexity that would be achieved with distributing the environment, an additional supporting functionality would be to develop a management console. This way an administrator should be able to easily detect any problems that might arise during typical runtime. This would be also helpful to manage maintenance on various aspects of the framework, such as the queues, database etc. A bit further down the road of improvement, when it comes to user interaction it might be useful if a GUI was added to the APF, as it was the case with Mule and BizTalk. This would make the framework even more user-friendly and easy to operate with. Thus a user friendly development environment would be offered that would ease the process of integration. Of course, these demands would be helpful later on in the process.

One can summarize that the routing and distribution of the components are essential to the next release, which would be based on the project that was presented in this document. It would be important to note that the configurability aspect of the APF needs to be preserved so that any additional integration would require fewer resources. In addition, the security integration would pose as an option for each of the customers’ needs. Some applications might not deal with sensitive data. Last but not least, the management console would add value for the administrators of the APF framework and the GUI consoles enhance the integration process with the APF.
References


Appendix A

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
exclude-result-prefixes="xs">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="channel">
    <rss version="2.0">
      <channel>
        <xsl:apply-templates select="title"/>
        <xsl:apply-templates select="link"/>
        <xsl:apply-templates select="author"/>
        <xsl:apply-templates select="description"/>
        <xsl:apply-templates select="item"/>
      </channel>
    </rss>
  </xsl:template>
  <xsl:template match="item">
    <item>
      <xsl:apply-templates select="title"/>
      <xsl:apply-templates select="link"/>
      <xsl:apply-templates select="author"/>
      <xsl:apply-templates select="description"/>
    </item>
  </xsl:template>
  <xsl:template match="title">
    <title>
      <xsl:value-of select="."/>
    </title>
  </xsl:template>
  <xsl:template match="link">
    <link>
      <xsl:value-of select="."/>
    </link>
  </xsl:template>
  <xsl:template match="author">
    <author>
      <Name>
        <xsl:value-of select="substring-before(text(), ' ')"/>
      </Name>
      <Last_Name>
        <xsl:value-of select="substring-after(text(), ' ')"/>
      </Last_Name>
    </author>
  </xsl:template>
  <xsl:template match="description">
    <description>
      <xsl:value-of select="."/>
    </description>
  </xsl:template>
</xsl:stylesheet>

Code Snippet 12: XSLT for transforming Provider to CMM formatted reply
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>My Feeds</title>
    <link>http://www.jcb.com</link>
    <author>
      <Name>James</Name>
      <Last_Name>Collin Brown</Last_Name>
    </author>
    <description>I am starting my rss feed service today</description>
    <item>
      <title>Day 1</title>
      <link>http://www.jcb.com/day1</link>
      <author>
        <Name>James</Name>
        <Last_Name>Collin Brown</Last_Name>
      </author>
      <description>RSS for dummies</description>
    </item>
    <item>
      <title>Day 2</title>
      <link>http://www.jcb.com/day2</link>
      <author>
        <Name>James</Name>
        <Last_Name>Collin Brown</Last_Name>
      </author>
      <description>Already an expert?</description>
    </item>
  </channel>
</rss>

Code Snippet 13: CMM formatted reply

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="channel">
    <rss version="2.0">
      <xsl:apply-templates select="title"/>
      <xsl:apply-templates select="link"/>
      <xsl:apply-templates select="author"/>
      <xsl:apply-templates select="description"/>
      <xsl:apply-templates select="item"/>
    </channel>
  </xsl:template>
  <xsl:template match="item">
    <item>
      <xsl:apply-templates select="title"/>
      <xsl:apply-templates select="link"/>
      <xsl:apply-templates select="author"/>
      <xsl:apply-templates select="description"/>
    </item>
  </xsl:template>
  <xsl:template match="title"/>
</xsl:stylesheet>
Code Snippet 14: XSLT for transforming CMM to Requester’s formatted reply

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>My Feeds</title>
    <link>http://www.jcb.com</link>
    <author>
      <Name>James</Name>
      <Last_Name1>Collin</Last_Name1>
      <Last_Name2>Brown</Last_Name2>
    </author>
    <description>I am starting my rss feed service today</description>
  </channel>
  <item>
    <title>Day 1</title>
    <link>http://www.jcb.com/day1</link>
    <author>
      <Name>James</Name>
      <Last_Name1>Collin</Last_Name1>
      <Last_Name2>Brown</Last_Name2>
    </author>
  </item>
</rss>
```
<author>
<description>RSS for dummies</description>
</author>

<item>
<title>Day 2</title>
<link>http://www.jcb.com/day2</link>
<author>
  <Name>James</Name>
  <Last_Name1>Collin</Last_Name1>
  <Last_Name2>Brown</Last_Name2>
</author>
<description>Already an expert?</description>
</item>

Code Snippet 15: Reply in Requester’s format

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="unqualified" elementFormDefault="qualified">
  <xs:element name="rss">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="title"/>
        <xs:element type="xs:anyURI" name="link"/>
        <xs:element name="author" type="authorCmplx"/>
        <xs:element type="xs:string" name="description"/>
        <xs:element name="item" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="title"/>
              <xs:element type="xs:anyURI" name="link"/>
              <xs:element name="author" type="authorCmplx"/>
              <xs:element type="xs:string" name="description"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Code Snippet 16: XSD for validation of the reply in the requester’s format