ATHABASCA UNIVERSITY

Enterprise Integration with Messaging

BY

Anuruthan Thayaparan

A thesis essay submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE in INFORMATION SYSTEMS

Athabasca, Alberta

February, 2011

DEDICATION

I would like to dedicate this work to my lovely wife and my best friend for all her support and encouragement throughout this MSc. Program and the writing of this essay.

ABSTRACT

Integration architects and developers are faced with many challenges when integrating disparate enterprise applications. First off, it is necessary to come up with the integration solution that meets organization's business needs. At the same time, they need to make sure that the solution requires minimum or no changes to the existing applications, so that cost of integration is minimal to the organization. This means the integration solution has to be technology neutral so that applications and systems built using different platforms and languages can be easily integrated. In addition, the systems integrated need to be loosely coupled so that changes to one system can be handled easily without impacting others. In order to tackle integration challenges like these, knowing and applying the appropriate integration styles and patterns is necessary. In addition, architects and developers need to be able to map different architectural design to appropriate technology and products. This thesis essay conducts a review on many of the integration architectures and patterns and how they help solve some of these challenges. It also reviews some of the technologies and products that are used to implement enterprise integration.

More specifically, the essay takes an in-depth look into enterprise integration with respect to messaging by reviewing various messaging architectures, patterns, and technology. So, there are three main objectives for this thesis. The first objective is to research and evaluate some of the common integration architectures. The second objective is to review and analyze various integration patterns, specifically messaging and interaction patterns, and discuss how they help solve many common integration problems. The third objective is to discuss current technologies and products that are being used to implement enterprise integration based on proven architectures and patterns.

TABLE OF CONTENTS

CHAPTER 1 1

INTRODUCTION 1

Purpose 1

Scope 1

Research Methodology 1

Thesis Essay Organization 2

What is Enterprise Integration ? 2

Integration Styles 3

What are Design Patterns ? 4

CHAPTER 2 6

INTEGRATION ARCHITECTURES 6

2.1 Point-to-Point 6

2.2 Hub-and-Spoke architecture 7

2.3 Bus architecture 8

2.4 Service Oriented Architecture (SOA) 9

2.5 Event Driven Architecture (EDA) 11

2.6 Enterprise Service Bus (ESB) 13

CHAPTER 3 15

MESSAGING PATTERNS 15

3.1 Synchronous messaging 15

3.2 Asynchronous messaging 15

3.3 Messaging Patterns 16

CHAPTER 4 22

INTERACTION PATTERNS 22

4.1 Common Interaction Patterns 22

4.2 Enforcing the rules of the Conversation 26

4.2.1 Orchestration 26

4.2.2 Choreography 27

CHAPTER 5 29

MESSAGING TECHNOLOGIES 29

5.1 Extensible Markup language (XML) 29

5.2 Web Services 29

5.3 RPC 30

5.4 REST 30

5.5 Middleware Technology for Messaging 30

5.6 Java Messaging Service (JMS) 31

CHAPTER 6 34

MESSAGING MIDDLEWARE PRODUCTS 34

6.1 Mule ESB 35

6.1.1 Mule ESB Messaging Architecture 35

6.2 TIBCO Middleware Products 38

6.2.1 TICBO Enterprise Message Service (EMS) 40

CHAPTER 7 42

CONCLUSIONS AND RECOMMENDATIONS 42

7.1 Conclusions 42

7.2 Recommendations 44

REFERENCES **45**

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the Athabasca University Faculty of Science and Technology at the School of Computing and Information Systems for preparing me to write this thesis essay and for providing me the access to online library to conduct this research. Special thanks goes out to Dr. Qing Tan for accepting the task of being the supervisor and providing me his guidance. Finally, I would like to recognize and offer my thanks to all the software professions and researchers who have published their work in the field of Enterprise Integration.

LIST OF TABLES

Table 1 - ESB products and their vendors 34

LIST OF FIGURES

Figure 1 - File Transfer 3

Figure 2 - Shared Database 3

Figure 3 - Remote Procedure Invocation 4

Figure 4 - Messaging Bus 4

Figure 5 - Point-to-Point Integration 6

Figure 6 - Meshed point-to-point connections 7

Figure 7 - Hub-and-spoke architecture with central message broker and adapters 8

Figure 8: Bus architecture with central messaging backbone 9

Figure 9 - SOA and its associated Entities 10

Figure 10 - Enterprise Integration with SOA and EDA 12

Figure 11 - Enterprise Service Bus and Service containers 13

Figure 12 - Synchronous vs Asynchronous Messaging 16

Figure 13 - Message Channel 17

Figure 14 - Message Router 17

Figure 15 - Message Translator 18

Figure 16 - Message Enricher 18

Figure 17 - Message Filter 19

Figure 18 - Message Splitter 19

Figure 19 - Message Router 19

Figure 20 - Message Sequencer 20

Figure 21 - Message Re-sequencer 20

Figure 22 - Message Endpoint 21

Figure 23 - Request-Reply 23

Figure 24 - Request-Reply with Retry 23

Figure 25 - Dynamic Discovery 24
Figure 26 - Subscribe-Notify 24
Figure 27 - Renewing Interest 25
Figure 28 - Reaching Consensus 25
Figure 29 - Web service interaction with Orchestration 27
Figure 30 - Web service interaction with Choreography 28
Figure 31 - Mule Architecture separated into three main layers 36
Figure 32 - Mule Endpoint configuration elements 38
Figure 33 - TIBCO Product suite 39
Figure 34 - Point-to-Point routing in EMS 40
Figure 35 - Publish-Subscribe routing in EMS 41
Figure 36 - Request-Reply interaction pattern in EMS 41

CHAPTER 1

INTRODUCTION

Purpose

The main purpose of this research is to review and analyze architectures, patterns, and technology that can be applied to solve common problems in enterprise integration. The essay should help integration architects and developers to tackle common problems and challenges that arise when integrating enterprise systems. In addition, the significance of this research is realized by the benefits of application integration to organizations. Some of the these benefits will be highlighted in this chapter. The end result of this research should yield both theoretical and practical foundation required to design and implement enterprise integration solutions.

Scope

As the discovery of integration architectures and patterns is an ongoing phenomena, the patterns reviewed in this essay are by no means a complete catalogue. The research may only cover a subset of the common and proven integration patterns and architectures that have been discovered by researchers and practitioners in this field. The computing platforms, languages and products described in this essay are used to simply illustrate how those architectures and patterns are realized in the implementation phase. Therefore, a review of complete set of the existing technologies and products used in integrating enterprise systems is not the goal of this essay. The aim is to try and map some of the common integration patterns and architectures to implementing technology. Although many integration styles are briefly reviewed, the primary focus throughout the essay is particularly on messaging.

Research Methodology

The research for this essay is primarily conducted by reviewing existing literature from peer reviewed articles, journals, documentations published by major middleware product vendors, and books authored by software professionals as well as researchers on the relevant subject matter.

Thesis Essay Organization

The thesis essay can be divided into two main parts. The first part of this thesis will focus on the theoretical and design aspects of Enterprise Integration. Some of the most proven Integration architectures and patterns have been categorized and described. Chapters 2, 3, and 4 are dedicated to the first part of the thesis. Chapter 2 will describe some of the common integration architectures. In Chapter 3, various messaging patterns are discussed and how they help solve common integration problems. Interaction patterns are discussed in Chapter 4. The use of interaction patterns assist with designing integrated work-flows that span multiple systems that are used within the context of a specific business process. The second part will focus on the implementation technologies in EI. It identifies some of the available technologies and products available that are used to implement integration solutions by leveraging those architectures and patterns discussed in the first part.

What is Enterprise Integration ?

Enterprise Integration (EI) is one of the important and inviting topics of research in the field of computing and information technology today. It is also one of the most challenging tasks as integration problems can occur at the business process level as well as at IT level. At the business process level, the integration problems occur when different modeling languages are used to represent the different views of an enterprise. At the IT architecture level, the problems occur when integrating two or more different applications [1]. In this essay, the focus is on how to solve integration problems that occur at the IT architecture level.

Business applications used in today's industries do not live in isolation. They need to work together to satisfy users' demands. Users expect quick and easy access to all business functions that an organization can offer. They do not want to be concerned about which system may provide a particular functionality. This requires disparate systems to be connected to each other into a larger solution and the required functionality exposed through a common interface. EI is about connecting disparate systems together within an enterprise or even systems across multiple enterprises, meaning that an organization may integrate with their business partners such as suppliers and vendors. It is essentially meant to solve the problem of data interaction and sharing between existing systems [2]. It ensures that different functional units in the organization are connected in a way that they can share information and business processes across the organization. This means that right people and the right processes have the right information at the right time. In other words, implementing EI should effectively increase productivity, reduce cost, eliminate data duplication, provide competitive advantage, and ultimately should result in increased profitability for the organization. So, the relevance of this research is realized in part by the benefits of EI to organizations in various industries. In addition, the research should also assist

integration developers and architects to tackle common problems and challenges that arise when integrating enterprise systems.

Integration Styles

Designing and implementing integration solutions can be very complex. There may be many possible solutions to the problem. Even after choosing what might seem to be the best possible solution, it will not typically be known until many months or even years after as new changes and additions to the the original architecture may be required. Over the years, application integration styles have evolved. Some of the earlier integration styles, but may still be appropriate in many systems today, are File Transfer, Shared Database, and Remote Procedure Invocation. Currently messaging is used more commonly as the choice of integration and leveraged by various integration architectures and solutions today.

a) *File Transfer -* In order to share data among different applications, the applications produce files that may be transferred to one or more different target applications to be consumed. In the middle, the files may get parsed and transformed by a separate application to fit the needs of the target system. This is also known as ETL (Extract, Transform, Load) process.

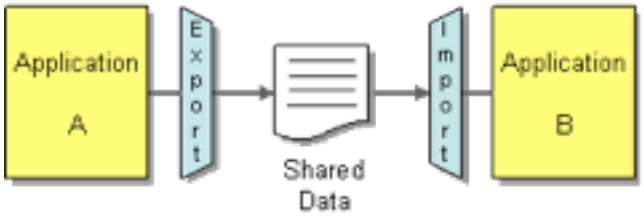

Figure 1 - File Transfer [17]

b) *Shared Database* - Integration can also be enabled by using a common database that can be used by multiple applications where the data is stored and consumed by those applications.

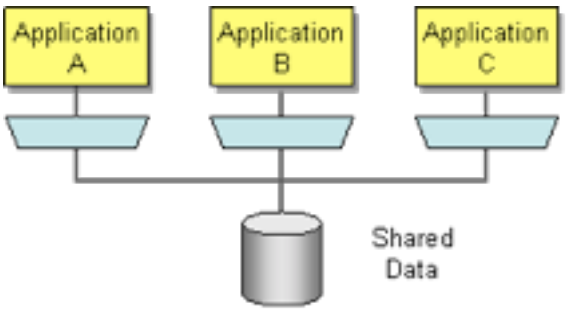

Figure 2 - Shared Database [17]

c) *Remote Procedure Invocation* - Applications expose some of their functionalities so that they can be invoked through remote procedure calls by other applications remotely. This enables exchange of data between those applications.

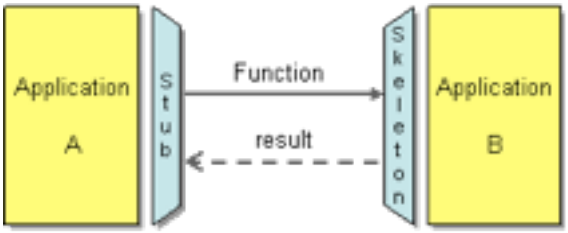

Figure 3 - Remote Procedure Invocation [17]

d) *Messaging* - In this thesis, the integration style in focus is Messaging. Based on messaging, several integration architectures and patterns will be discussed. They can guide solution developers to design and architect EI that enable data sharing among different applications by simply exchanging messages. The integration using messaging is usually achieved through the use of some form of Middleware technology and products. The Middleware provides the mechanism and the platform to enable data transport, data transformation, routing, and so on.

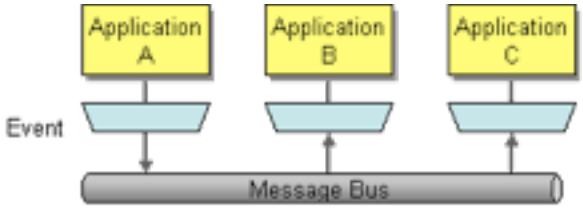

Figure 4 - Messaging Bus [17]

Before discussing the various design patterns used to develop EI solutions, the next section provides some background on design pattens first.

What are Design Patterns ?

In software engineering, design patterns are known as solutions to commonly occurring problems. Patterns are an abstract representation of an architecture, in the sense that they can be realized in multiple concrete forms [3]. For example, the publish-subscribe pattern describes an abstract mechanism for loosely coupled integration and many-to-many communications between publishers of messages and subscribers who wish to

receive those messages. Patterns do not use specific technologies, as they focus on the architectural concepts and not the implementation technology. These patterns are captured from experts' knowledge in the fields such as object-oriented design and enterprise application architectures. As there is no "one size fits all answers" to the challenges and problems in these fields, each pattern handles a specific design problem, describes the problem, discusses the design considerations, and presents an elegant solution to solve the problem. According to James Noble, "Patterns are crucial to the art and science of software design and programming, rooted in hard-won practice and experience" [4]. Since patterns are derived from real-life practice, each pattern incorporates the experience gained by senior developers and architects from repeatedly building solutions and learning from their mistakes. Therefore, the patterns are categorized and documented some time after they have been practiced and proven in the industries. The patterns discussed in this report are by no means complete as there are new patterns being discovered as part of experts' daily job facing different challenges in different customer facing environments and their projects.

One of the very first books on software design patterns was published in 1995. This book was titled, Design Patterns: Elements of Reusable Object-Oriented Software, and was written by Eric Gamma et al [5]. This book has actually been known to bring design patterns into mainstream. In 2002, Martin Fowler published a book titled, Patterns of Enterprise Application Architecture [6]. This book mainly covered patterns collections that focused on enterprise application architecture. The patterns presented in this book described how to layer an enterprise application, how to organize domain logic, how to incorporate domain logic to a relational database, how to design a web-based presentation, and some important principles in distributed design. A catalog of application architecture patterns can be viewed at http://martinfowler.com/eaaCatalog.

In 2003, Gregor Hohpe teamed up with Bobby Woolf to write a book on Enterprise integration patterns. This book titled, "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", describes integration solutions across many implementation technologies [7]. So, the book actually goes beyond just teaching the patterns, but also maps them to some of the available technologies. It describes 65 distinct integration patterns with consistent vocabulary and visual notations. This book also plays an important role in the research for this report as it serves as reference to many of the patterns discussed in this report.

CHAPTER 2

INTEGRATION ARCHITECTURES

Integration architectures are at the highest level of abstraction in the process of coming up with an EI solution. In this chapter, some of the most common architectures used in EI will be discussed, starting from the earliest to the ones that are more commonly used today.

2.1 Point-to-Point

The most simplest approach to connect one system to another would be to connect them directly with each other through a custom interface. This allows the two systems to communicate with each other using an API that is custom designed and developed to handle data transfer between them. This approach is commonly referred to as Point-to-Point integration.

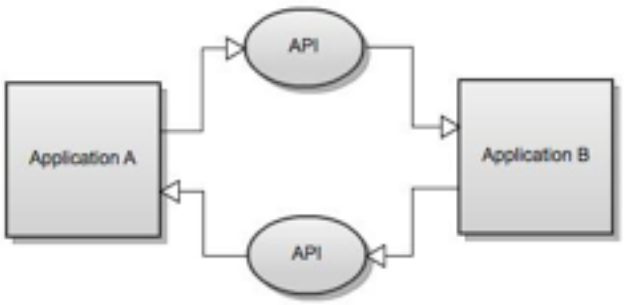

Figure 5 - Point-to-Point Integration

However, there are several potential problems with this approach when it comes to integration of multiple systems across a large enterprise. With just two applications, it is relatively simple to maintain as there is just one API code with a single predefined communication standard and interface that need to be known to the two applications. When more applications need to be integrated and need to share data with each other, more details need to be maintained for each connection. The number of connections needed can be calculated using the following formula.: n (n-1) / 2. The variable n is the number of applications that need to be connected in a fully meshed point-to-point connections. For example, ten systems would require 45 point-to-point connections (10

* 9 / 2 = 45). The number connections increase significantly as more applications need to be connected.

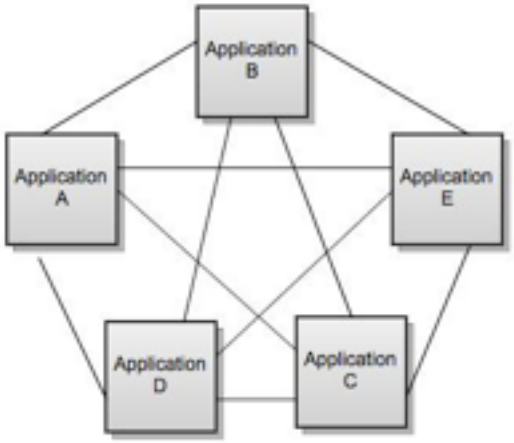

Figure 6 - Meshed point-point connections

Each application needs to know about how to connect to each other. Each application needs to know how to transform/translate data formats between each other. If multiple applications need to be involved in completing a single request, this also means all participating applications need to be active and available to handle the request at all time. Since applications are tightly-coupled with each other, it also becomes difficult to manage changes. A change to one application's interface affect all others. So, this approach quickly becomes very complex and difficult to maintain.

2.2 Hub-and-Spoke architecture

Hub-and-spoke architectures is one of the ways used to solve the problem in point-to-point architecture. In this architecture a single centralized broker, referred to as the "hub", handles the message exchange, transformation, and routing for the systems that need to be integrated. These systems are integrated by connecting them to the broker through their adapters called the "spokes". Each spoke connecting the source system and the hub converts the message format to the format that the hub understands. The hub then takes care of transformation and translation of incoming data in a way that the target system can understand. Figure 7 below illustrates the Hub-and-spoke architecture.

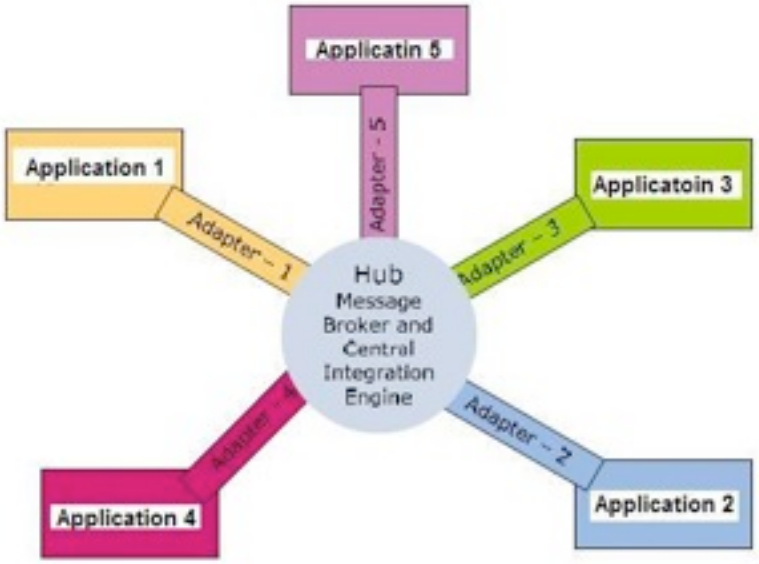

Figure 7 -Hub-and-spoke architecture with central message broker and adapters [8]

By having a central hub makes this architectural solution relatively easier to manage. However, the scalability becomes dependent on the hardware. To overcome this hardware limitation, many vendors have come up with federated hub and spoke architectures in which multiple hubs can be put in place to scale better. Any changes to rules such as transformation or routing in one hub would get propagated to other hubs automatically. As this architecture provides central management of all hubs, it also helps to bring down support cost.

2.3 Bus architecture

The Bus architecture has a central messaging backbone that is referred to as the message bus. The messages are propagated though the bus from source systems to target systems. Messages from the source systems get sent to the bus using the system adapters and they flow to the target systems using this message bus. The message transformations take place in source adapters into a format required by the bus. Similarly, the messages are transformed in the target system adapters to the format that the target systems can use. The main difference between hub-and-spoke and bus architectures is in the integration engine. For the bus architecture, the integration engine that performs message transformation and routing is distributed in the system adapters as seen in Figure 8 below. This actually scales better than the hub-and-spoke design because the adapters have integration engine and run on the same platform on which source and target systems run. This makes it less complex to

manage than maintaining the hub.

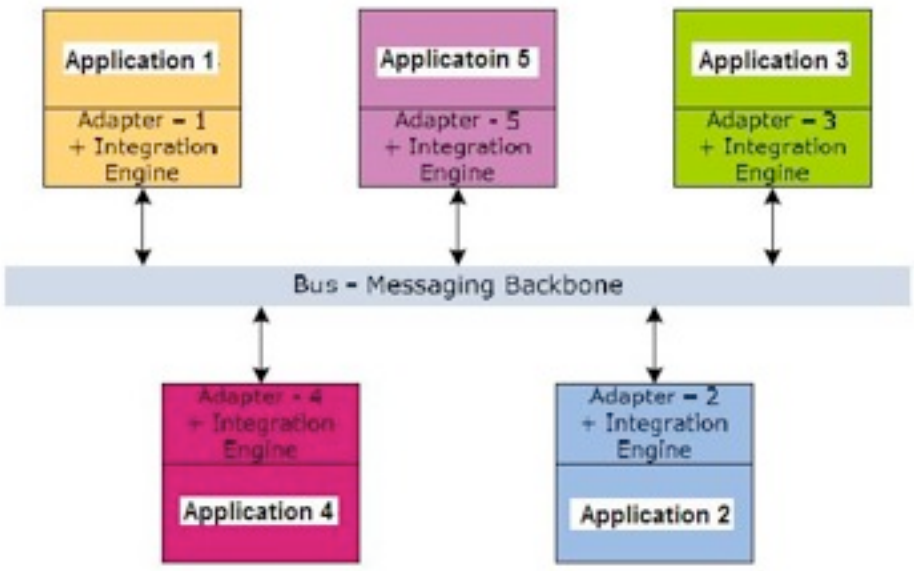

Figure 8: Bus architecture with central messaging backbone [8]

2.4 Service Oriented Architecture (SOA)

Service Oriented Architecture is one of the most talked about integration architecture or solution in the market within the last decade. Before SOA came into mainstream, Enterprise Resource Planning (ERP) was another software integration solution that was popular in 1990s. However, the problem was that in order to implement ERP, most companies had to first reengineer their business processes to adopt ERP standard business processes [9]. For most organizations, this made it impossible to implement ERP as their current business scheme was not compatible with the standard required by ERP. SOA on the other hand does not require the business to change their standards or move away from the existing legacy systems that organizations rely on [9 35]. SOA is the notion of building the infrastructure that enables loosely-coupled interoperability between disparate systems using discoverable services and messaging. These systems may be of different types of technology and platforms. A service is a specific functionality or operation exposed through a common interface by the source system. Each service instance interacts with disparate systems or other services through a loosely-coupled message-based communication model. The most basic pattern that is implemented in SOA interaction is referred to as “Request-Reply” pattern. Service consumer sends a request to the service provider. The service provider replies to the request in response. In addition, SOA consists of various entities configured together to support the *find, bind, and execute* paradigm as shown in Figure below [10]. The entities involved are as follows.

a) *Service provider and consumer* - The service provider is the system that receives requests from the service consumer. Before any service consumer can request the service, the service provider needs to publish its service contract in the service registry for access by service consumers. The service consumer may be an application, another service, or some other type of system that requires a service. The service consumer locates the service in the service registry, binds to the service over a transport, and executes the service request by sending a message in a format according to the service contract. The service provider executes requests and responds to the consumer in the format specified in the service contract.

b) *Service Registry* - The Service Registry is a network-based directory that accepts and stores service contracts from service providers. It then provides those contracts to the service consumers. It is essentially a central repository that contains all available service contracts that can be retrieved and thus enables the services to be dynamically discovered by the interested consumers.

c) *Service Contract* - The Service Contract is the specification of the way a service consumer and service provider can interact with each other. It specifies the format of the request and response from the service provider. It may also specify quality of service levels and specifications for the non-functional aspects of the service.

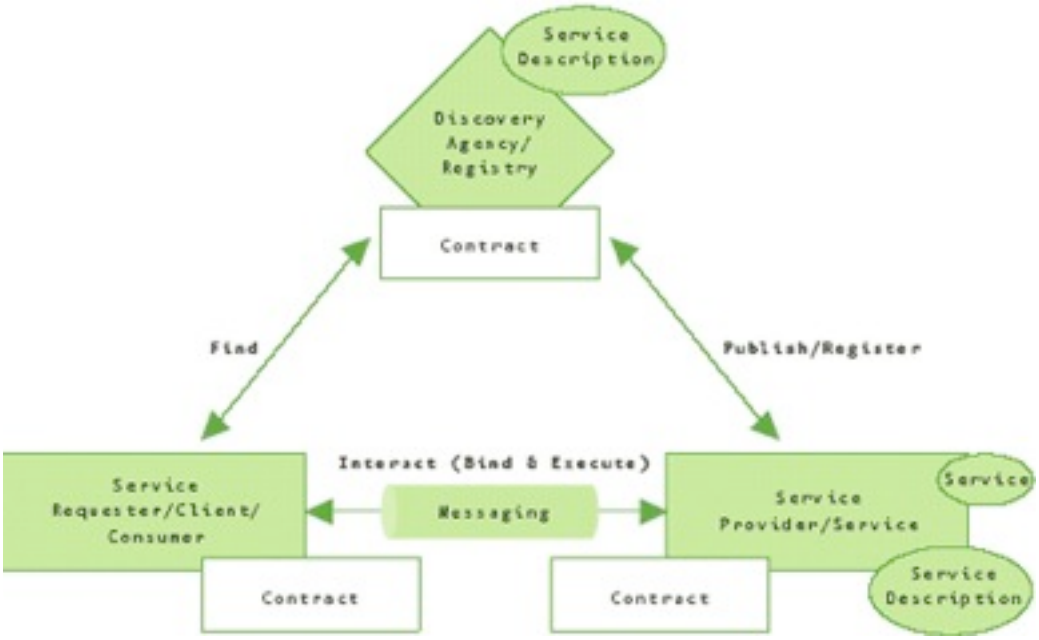

Figure 9 - SOA and its associated Entities [10]

SOA basically enables enterprises to be organized around services instead of around applications. But, SOA alone is not enough to successfully implement EI. In a given SOA implementation, each service may have a predefined business objective. At the same time, multiple services may need to interact with each other to execute a particular business process. Each business process may consist of various business activities in a complex workflow. This means transaction management is required in cases where one of the services fails in the workflow and thus needs to be rolled back. Therefore, SOA requires intermediaries such as the message broker or the bus to coordinate the service interaction. Enterprises also need integration built around business events that are triggered during a workflow. Various business events would then need to trigger service interaction among different systems. So, services implemented in SOA also need to react to business events. This is handled by building Event Driven Architecture as discussed next.

2.5 Event Driven Architecture (EDA)

In EDA, interaction between systems is triggered in response to an event. An event is simply a change in state in a system or something important that happens in a business. So an event in the business or system may be a new problem, a new opportunity, a threshold, or some sort of significant deviation that has an impact to the business [11]. This event would need to be sent to another system so that proper action can be taken in response. So, the fundamental interaction pattern that is implemented in EDA is referred to as “publish-subscribe” pattern. The event is published and the subscribers consume and react to the event. For example, inventory system may trigger an event that the store is running low on supply and publish an event message indicating this problem. Subscribing ordering system can pick up this event message and submit an order for the specific product that the store needs. The ordering process may be yet another event that is published so that the subscribing supplier system can pick up this order request event and fulfill the order. When EDA is implemented with SOA, an event can trigger the invocation of one or more services. Those services may perform simple functions, or entire business processes like the order processing example. This interaction between events and services is commonly referred to as event-driven SOA [11,12]. Figure 10 below depicts how SOA and EDA fit into an integration solution together.

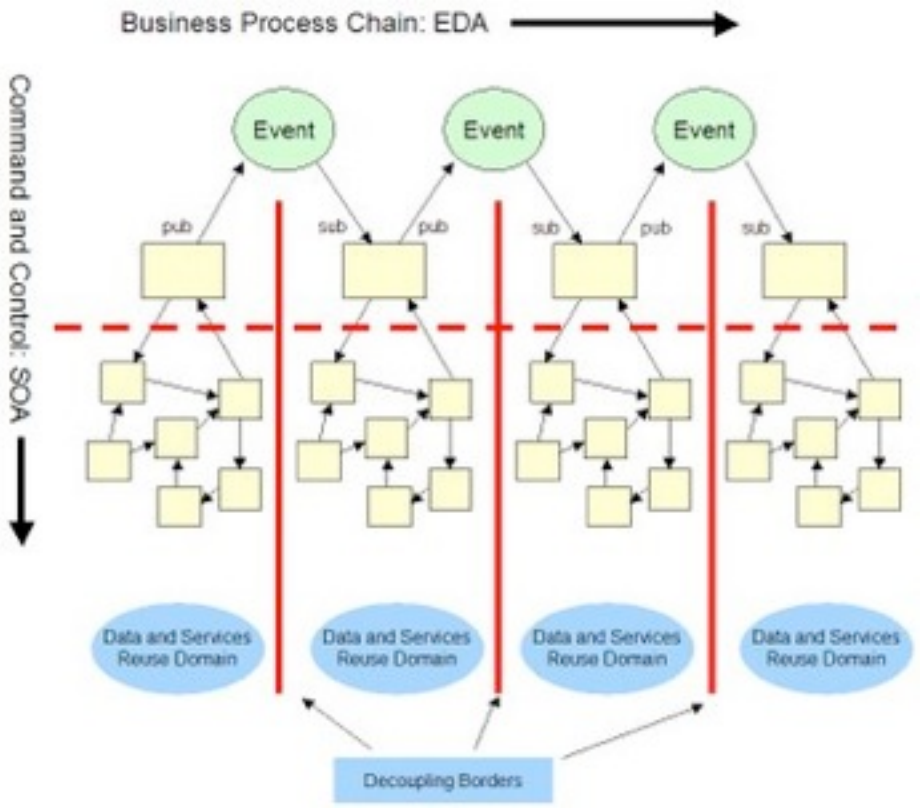

Figure 10 - Enterprise Integration with SOA and EDA [12]

As shown in Figure 10 above, EDA is illustrated as a series of business events that are published and consumed by different systems in business process chain. This is commonly referred to as the publish-subscribe interaction pattern. A set of interacting systems may reside in different domains, where each domain has its own SOA implemented as their systems' functionalities are exposed as services. This is referred to as the request-reply interaction pattern.

The architectures discussed so far can actually work with each other to implement an integration solution as a whole. For example, SOA can be implemented with EDA to implement an event driven service interaction. However, in order to implement these architectures, a form of intermediary such as a messaging middleware is needed to facilitate the interaction that is required. So, a messaging engine or broker along with event-driven service interactions can be implemented using either the Bus architecture or the Hub-and-Spoke architecture that were discussed earlier. One such solution is more commonly realized by using an Enterprise Service Bus (ESB). ESB has become one of the most popular intermediaries today that provides the infrastructure for implementing SOA and EDA type of integration solutions to enterprises. It is considered as the most common and effective middleware solution for Enterprise Integration today.

2.6 Enterprise Service Bus (ESB)

ESB is an application integration model that is based on the Bus architecture. It effectively extends the Bus architecture by providing a scalable infrastructure to facilitate architectures such as SOA and EDA. Many vendors today use the term ESB in their middleware products and provide the APIs which can be used to develop web services and make those services interact with each other reliably. According to Chappel, “An ESB is the implementation backbone for a loosely coupled, event-driven SOA that enables a highly distributed universe of named routing destinations across a multi-protocol message bus” [13]. ESB can scale beyond the limits of a hub-and-spoke architecture as the systems can be integrated using adapters that run on the same platform on which source and target systems run. The term ESB is now also used to describe an integration solution or product sold by various vendors to implement enterprise integration [14]. It is a combination of a middleware technology and products that helps to implement enterprise integration as it mediates the interaction using messaging between disparate systems that are loosely-coupled software modules or services. Business events can be propagated through the service bus in the form of messages.

Figure 11 below illustrates ESB as an integration module that connects various systems through the use of service containers. Different service containers in an ESB expose message-driven interfaces for the purpose of sharing data between applications, both synchronously and asynchronously. A service container can provide a number of facilities for the service implementation such as event dispatch, thread management, security through reliable message delivery [13].

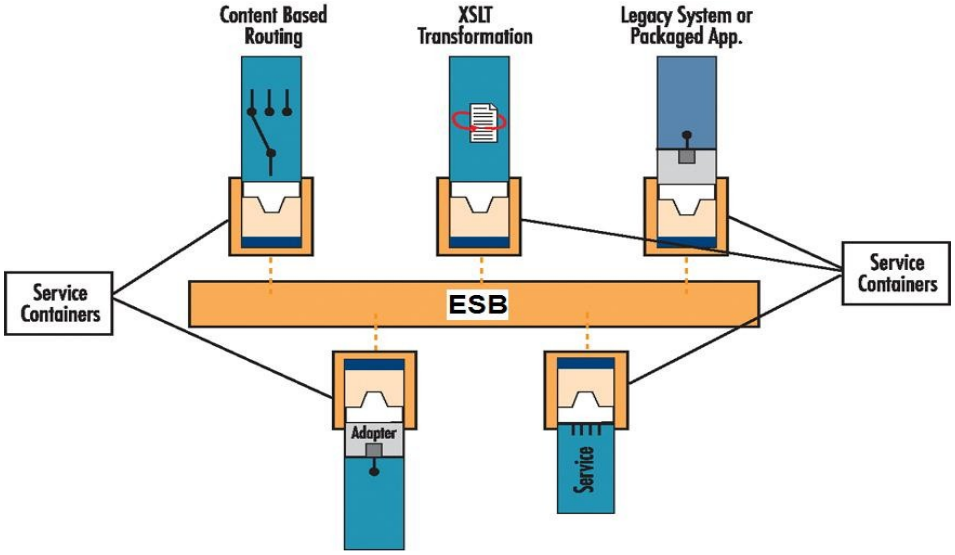

Figure 11 - Enterprise Service Bus and Service containers [13]

ESB effectively needs to be able to support three major styles of enterprise integration. Firstly, it needs to support SOA in which applications communicate through reusable services. Secondly, since those service-oriented interactions leverage underlying messaging and event communication models, ESB needs to support message-driven architectures in which applications send messages through the bus to receiving applications. Lastly, it also needs to facilitate event-driven architectures in which applications generate and consume messages independently of one another [15]. Chapter 6 will revisit ESB with respect to various ESB products offered by different vendors.

Chapter 3 and 4 extend the discussion from this chapter by taking a closer look at some of the messaging and interaction patterns used to implement EI solutions around SOA and EDA. It is apparent that messaging underpins the required interaction when integrating disparate systems. So, Chapter 3 describes messaging in more detail and some of the common messaging patterns that help solve many of the common integration problems. Chapter 4 describes interaction patterns. These patterns are used in having multiple systems work together towards a common goal. These interactions can span within a single enterprise or multiple enterprises such as suppliers, manufacturers and vendors to carry out various business processes.

CHAPTER 3

MESSAGING PATTERNS

Messaging underpins some of the most proven and desirable integration architectures and solutions described in chapter 2. A message is simply a packet of data that is transmitted from one system to another. Each message, at minimum, consist of two parts:

a) *Header* - describes the message being transmitted and contains information such as its source, its destination, and other data that can be used by messaging system to route the message.

b) *Body* - consists of the content of the message being transmitted.

Simply put, application integration is enabled by having the applications communicate with each other and share data through sending each other messages. In order to integrate these systems, a messaging system is put in the middle, referred to as the Middleware, to coordinate the messages that need to be passed from one to the other. Having the messaging system in the middle enables loose coupling among disparate systems because each system communicates through the messaging middleware and does not need to know specifically about each other when sending and receiving messages. This section starts by describing two fundamental types of messaging: Synchronous vs Asynchronous messaging. It then describes some of the common asynchronous messaging patterns.

3.1 Synchronous messaging

In synchronous messaging, the calling process makes a call and it blocks its process until the called process completes its operation and the response is received. This means both parties (source and target systems) have to be available for the operation to be successful. In other words, both systems have to be tightly coupled when it comes to synchronized messaging, and may not be possible for all systems in an enterprise to be integrated this way. However, some systems may actually require the communication to be synchronous depending on the business requirement.

3.2 Asynchronous messaging

In asynchronous messaging, however, the caller's process is non-blocking as it only initiates the operation. Asynchronous messaging architectures have become most common and proven ways to integrate enterprise systems as they allow for loosely coupled solution that overcomes the limitations of remote communication, such as latency and unreliability. The concept of asynchronous messaging is actually very much similar to how interactions occur between people in real life situations from different physical locations. For example, people may communicate with each other through

phone calls. Person A calls person B. Person A cannot always expect Person B to answer the phone call, because Person B may not necessarily be available to pick up the phone. So, person A may have to just leave a voice mail to provide information or instructions to do something and expect a response back. Also, person A cannot assume that Person B will receive the message and respond back within specific timeline . So, if person B does not respond within a certain amount of time, person A may try to call person B again. This is exactly how the situation would be between different systems (system A and system B) through asynchronous messaging. One cannot assume the status of other systems at any given time. Similar to person A and B, system A cannot assume that system B will respond to a request at any given time. As we live in an asynchronous world, asynchronous messaging allows us to face the reality of uncertainty in real life. By reducing or eliminating those assumptions, systems can be loosely coupled with each other using asynchronous messaging. Figure 12 below illustrates difference between synchronous and asynchronous messaging. Next, some of the common patterns used in asynchronous messaging systems will be discussed.

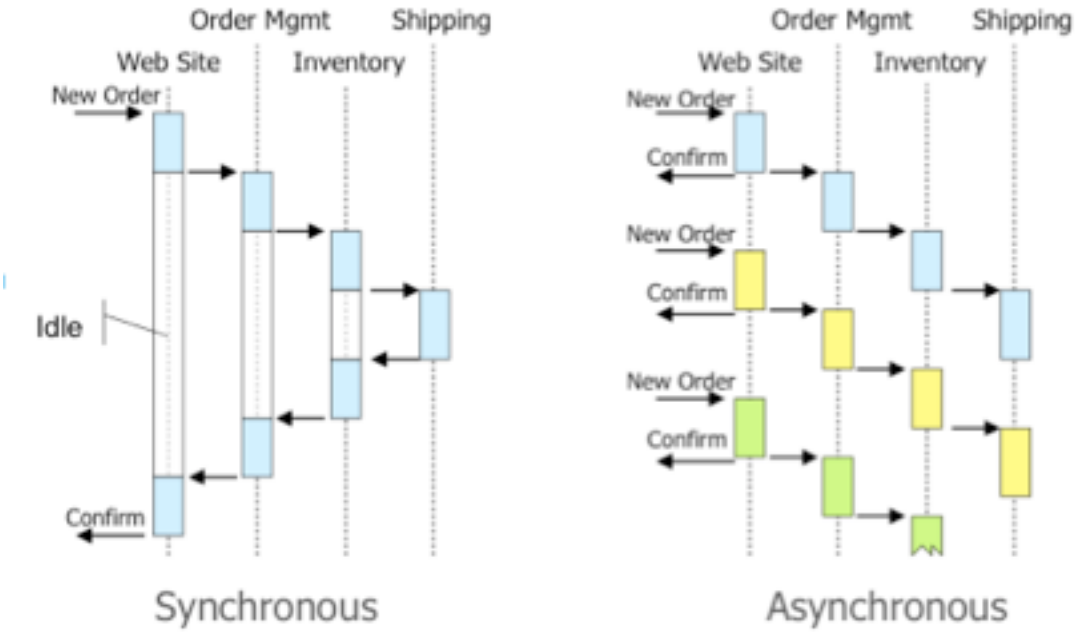

Figure 12 - Synchronous vs Asynchronous Messaging [16]

3.3 Messaging Patterns

Asynchronous Messaging Patterns effectively assist with designing asynchronous messaging solutions. These patterns exist at different levels of abstraction. Some patterns are used to represent the message itself, or attributes of a messaging system.

Others are concerned with the flow of messages from a source system to one ore more target systems. They describe various ways a message can be directed from one system to the other by using complex mechanisms. They are also used to represent creation of message content or change the content of a message. The following are some of the most commonly known messaging patterns.

a) *Message Channel* - A Messaging system transmits messages through a Message channel that connects the sender application to the receiver application. A Message channel can be described as a virtual pipe established from one application to the other. It may also be referred to as message Queue or Topic based on the type of channel it is and the technology or vendor used to implement the message channel. The difference between Queue and Topic based message channel will be described in chapter 6.

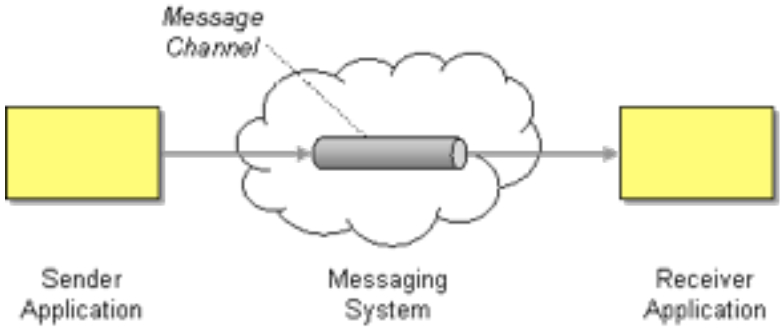

Figure 13 - Message Channel [17]

b) *Message Router* - The primary function of a Message Router is to connect one Message Channel to multiple Message Channels. It does not modify the message content as it is only concerned with the message destination. It can use any number of criteria to determine the output channel of an incoming message. This allows for the decision criteria of messages to be maintained in a single location. This pattern also enables the publish-subscribe interaction pattern as it allows the messages published by a source system to be routed to multiple subscribing applications.

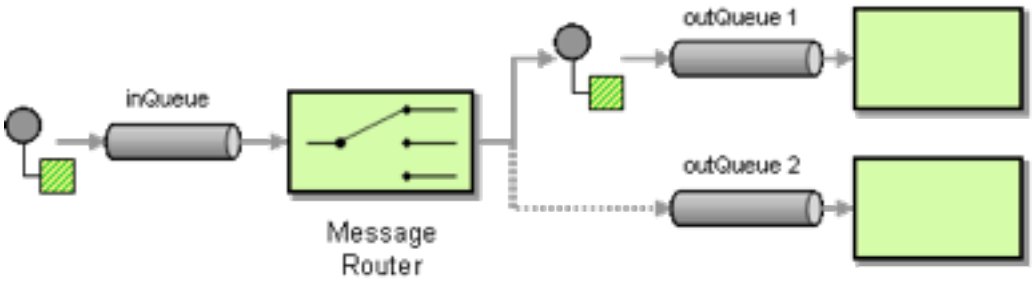

Figure 14 - Message Router [17]

c) *Message Translator* - The Message Transformation is concerned with the message content itself as it acts as a message translator. It is usually necessary to translate the

messages from one format used by the source system to another format that is required by the target system. One of the benefits of being able to transform the messages is that it allows the applications to be decoupled from each other. When integrating existing systems, there is no need to make any changes to any of the systems in order to adhere to another system's message format. The translator on the messaging middleware can do the job instead.

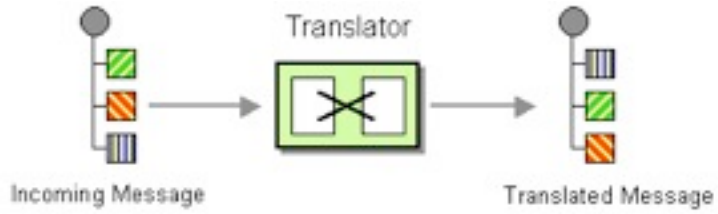

Figure 15 - Message Translator [17]

d) *Message Enricher* - Messages Enrichment is necessary when the message content needs to be enriched by adding additional data to the incoming messages before being routed to its destination. The additional data, for example, may need to be retrieved from a database and appended to the message content.

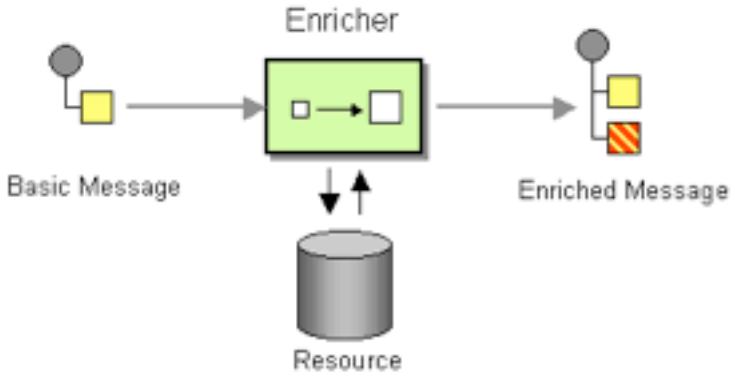

Figure 16 - Message Enricher [17]

e) *Message Filter* - The Message Filter is a way for the target system to avoid receiving unwanted messages from the source system. It is similar to a Message Router but with a single output channel. If the content of an incoming message matches the criteria specified in the Messages Filter, it is routed to the output channel where the target system would be able to pick up the message. Otherwise, the messages is discarded.

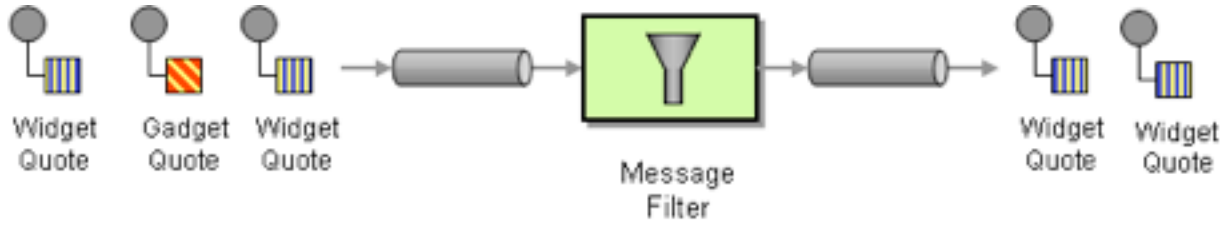

Figure 17 - Message Filter [17]

f) *Message Splitter* - The Message Splitter is useful when a message contains multiple elements that have to be split and processed in a different way. The Splitter would essentially break the single message into multiple messages, each containing data related to one element.

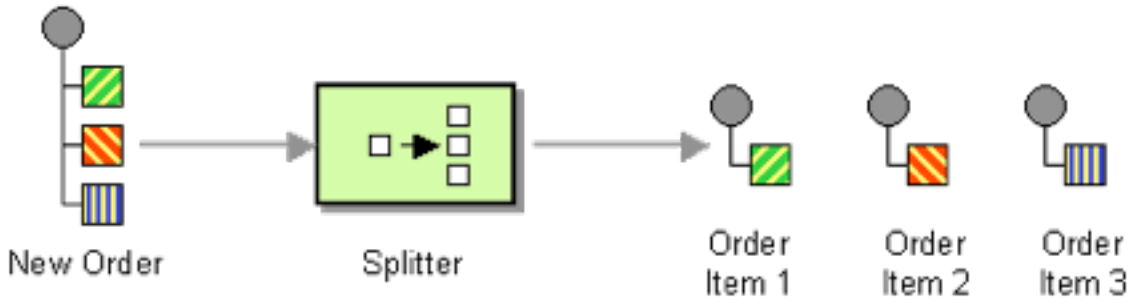

Figure 18 - Message Splitter [17]

g) *Message Aggregator* - The Messages Aggregator is useful when multiple but related messages have to be combined into one so that it can be processed as a whole. It would need to receive a stream of messages, identify the messages that are related, and publish a single aggregated message to the output channel.

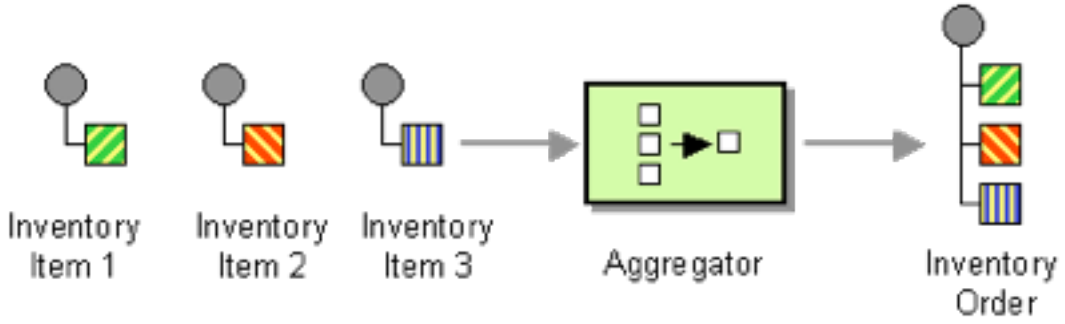

Figure 19 - Message Router [17]

h) *Message Sequencer* - Message Sequencer needs to be implemented when large amount of data need to be transmitted but is too big to be transmitted as a single message. The Message sequencer breaks up the large chunk into message-size chunks, mark each with sequence identification fields, and send them. There are three Message sequence identification fields as follows: a) Sequence identifier that distinguishes the broken up chunks as cluster of messages from others, b) Position identifier sequentially orders the messages in sequence, and finally c) the End indicator marks the final message in the cluster and specifies the number of total messages in that cluster.

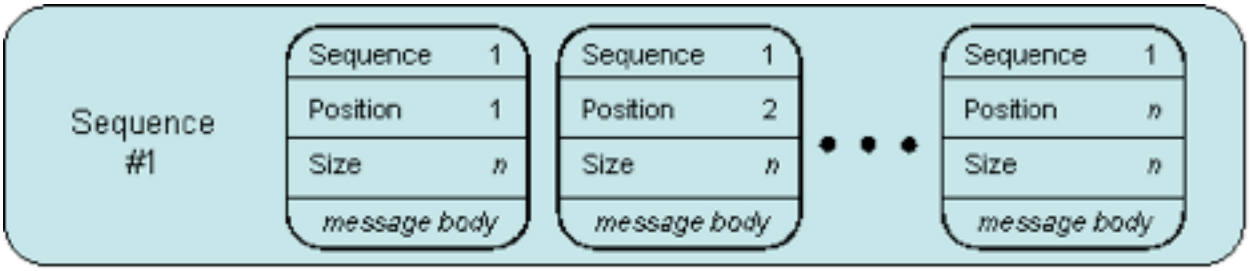

Figure 20 - Message Sequencer [17]

i) *Messages Re-sequencer* - Messages may get out of sequence when they have to be processed differently or by different components. When this occurs, Message Re-sequencer can be used to put the individual messages that are out of sequence back into the correct order. In order to do this, each message needs to have a unique sequence number. These sequence numbers help the re-sequencer to detect when they are out of order. When a messages with the higher sequence number arrives before the message with a lower sequence number, the Re-sequencer has to store the messages with the higher sequence number until it receives all the messages with the lower sequence numbers. When the buffer contains all the consecutive sequence numbers, the Re-sequencer would send the messages to the output channel and then remove them from the buffer.

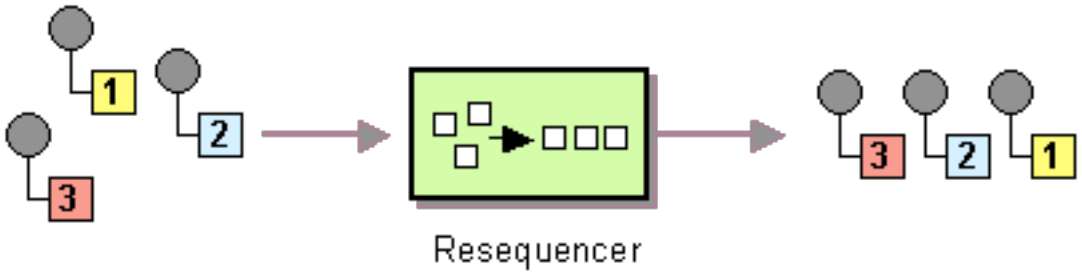

Figure 21 - Message Re-sequencer [17]

j) *Message Endpoint* - Applications and the messaging system need a way to communicate with each other because they are separate entities. A message endpoint enables them both to connect to each other. A sending application can use a message endpoint to send and receive messages to messaging system and the messaging system can then route and dispatch messages to the message endpoint of the receiver application. This way, the sending and receiving applications are decoupled from each other as they don't need to know about each other and only need to interact with their message endpoints. Any changes to one application does not have an impact on other applications as long as its endpoint gets updated accordingly.

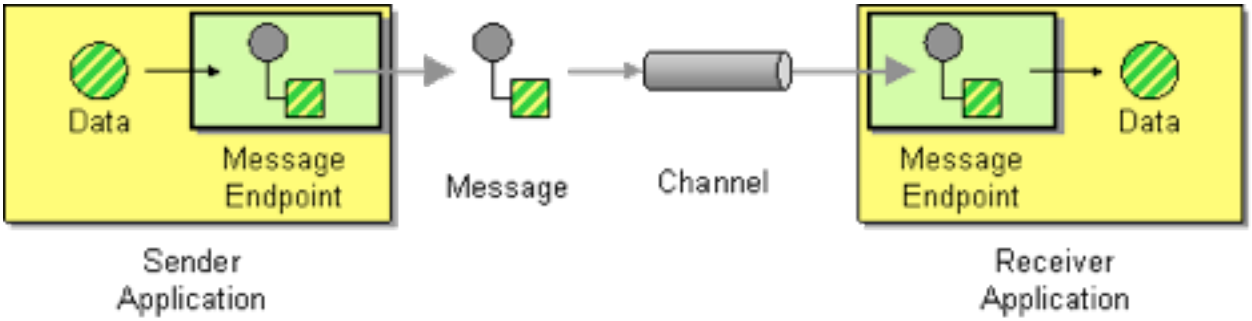

Figure 22 - Message Endpoint [17]

CHAPTER 4

INTERACTION PATTERNS

In chapter 3, patterns related to messages were discussed. They described patterns on how messages could be routed, transformed, enriched, and so on. In other words, those patterns are concerned with message content, message format, and message delivery. However, EI needs to be concerned with much more than how a message should be sent from system A to system B. In EI, systems need to communicate with each other by passing messages and responding to each others' messages. So, this chapter focusses on patterns that describe various ways that systems can interact and respond to each other using messaging. There may be complex interactions that typically need to take place when different systems need to be integrated to provide their services or carry out various activities in a complex work-flow of a business process. Holpe refers to such patterns as *Conversation Patterns* [18]. Various interaction patterns will be described in section 4.1 below.

When two or more systems communicate and share data with each other, the type of communication may take on the form of a conversation between each other. The conversation patterns describe and illustrate the types of interactions that can occur when two or more systems are integrated in order to share data or carry out a specific function. They essentially assist with designing communication based on some rules among multiple systems using messaging. For example, when systems expose parts of their functionality as Web Services, a particular business process may actually take several Web service interactions to complete. This means there are multiple steps in the conversation that need to take place based on specific rules applied to the conversation.

4.1 Common Interaction Patterns

a) *Request-Reply* - Request-Reply is the most simplest form of conversation between two systems. System A sends a message to System B. The message may be to send instructions or share data. System A may wait and block until the response from System B is received (Synchronous) or continue to perform other tasks (Asynchronous) while System B processes the request and responds. So, Request-Reply may be implemented synchronously or asynchronously. However, as mentioned before, asynchronous communication has the advantage over synchronous communication due to decoupling of systems being integrated. A real-life example of this pattern would be when a user submits a purchase order online through a web browser (request), the user needs to receive a order number and confirmation of purchase (response).

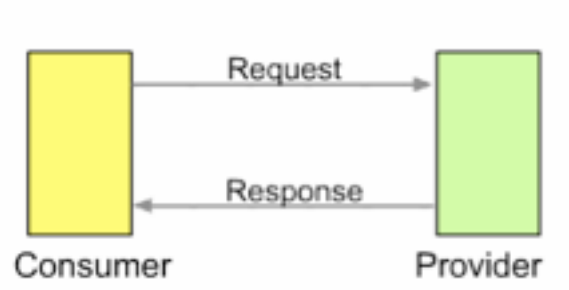

Figure 23 - Request-Reply [18]

b) *Request-Reply with Retry* - As an extension to the Request-Reply pattern, the sender may need to repeat a request until a response is received or repeat it n times until it gives up. It is important that provider or the receiver needs to be idempotent. In other words, sending the request one time or multiple times should have the same effect on the receiver. So, retrying the requests multiple times should not have any unwanted effect [18].

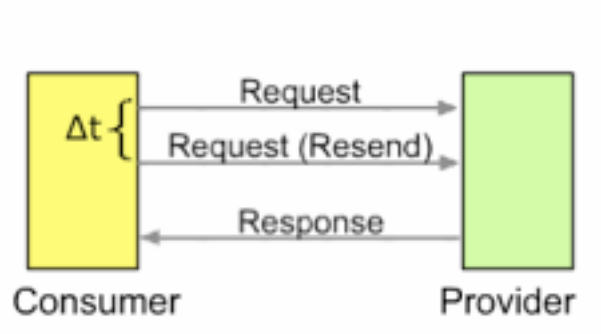

Figure 24 - Request-Reply with Retry [18]

c) *Dynamic Discovery* - In Dynamic Discovery pattern, the system sends a broadcast request to a channel where multiple target systems may be listening on. This form of requests to multiple target systems is also known as the publish-subscribe pattern. In this case however, the providers are the subscribers of this channel and they would decide whether or not to respond to the request. So, the requester would then need to choose which provider to respond back to. After the interested providers send their responses, the requestor chooses the desired provider from all the responses. The requestor then initiates interaction with the chosen provider [18]. A real-life scenario for this pattern would be when a user requests a quote for auto insurance on a web portal, the backend service fires off requests to multiple insurance providers and renders each of the response back to the user. User can then choose the one that he/she prefers and starts the application process.

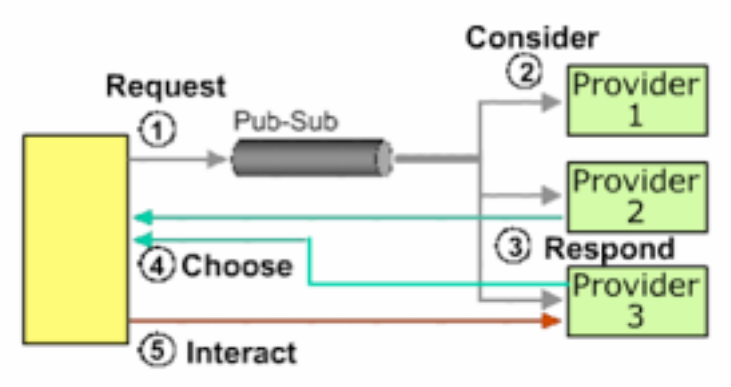

Figure 25 - Dynamic Discovery [18]

d) *Subscribe-Notify* (Multi-responses) - In this pattern, the subscriber expresses interest in receiving notifications and receives messages until the subscriber sends a stop request. Provider notifies subscriber of end of transmission. Examples: WS-Eventing, WS-Notification [18].

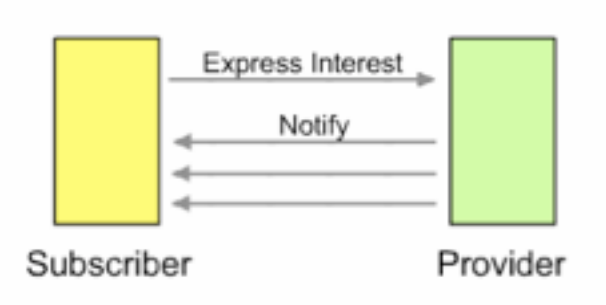

Figure 26 - Subscribe-Notify [18]

e) *Renewing Interest* - The Renew Interest pattern is also referred to as the "Lease" model or the “Magazine Model” [18]. In the “Lease” model, the subscriber has to renew actively after certain interval. Otherwise, the subscription would automatically expire. In the “Magazine Model”, the subscriber does not need to know when to send a renewal request, because the provider manages the state of each subscriber. Before the next expiration, the provider sends a request to the subscriber for renewal after a certain time interval.

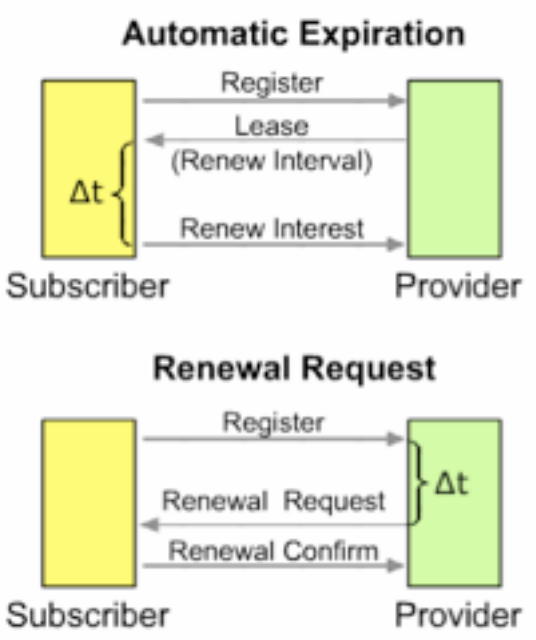

Figure 27 - Renewing Interest [18]

f) *Reaching Consensus* - In this pattern, the conversation is coordinated. The coordinator initially sends the request and then tracks responses from each participant. Based on the responses, the coordinator makes the decision and then broadcasts the decision. A simple real-life example for this would be the type of communication that occurs when someone is assigned to coordinate when and where to go out for lunch at the office. The coordinator sends everyone an email asking when and where they would like to go out for lunch. After everyone responds, the coordinator makes the final decision and sends the notification to everyone with the details on when and where to go out for lunch together [18].

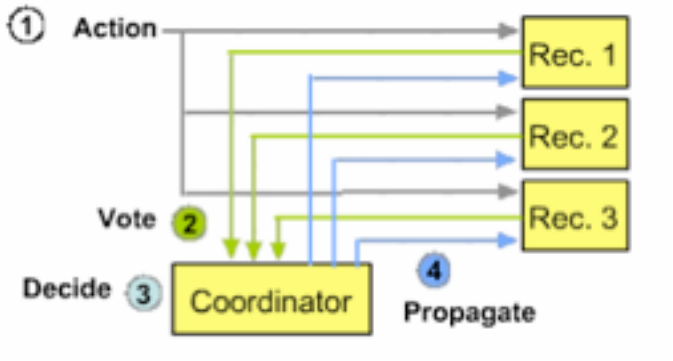

Figure 28 - Reaching Consensus [18]

Several more service interaction patterns have been discussed by Barros et al in [19] that involve interactions among more than two systems. The authors refer to these patterns as multi-transmission interaction patterns. One of them is “Contingent request” pattern in which, system X makes a request to another system Y. If X does not receive a response within a certain timeframe, X alternatively sends a request to another party Z, and so on [19]. Next section will discuss ways to enforce such rules to the interaction or conversation.

4.2 Enforcing the rules of the Conversation

When some of the more complex conversation patterns such as “Dynamic Discovery” or “Contingent request” are implemented, the conversation among the systems needs to have specific rules enforced in order to execute the workflow of activities and messages exchanged during the conversation. Without these rules, systems would not know how to carry out the conversation. For example, a system needs to know whether to send a message once and forget it, or send a message and retry after specified time has elapsed if no response is received. It essentially needs to know what to do when the response is received. Based on the type of response, it may need to send a message to another system or call some other internal process. Simply put, in order to make disparate systems or services interact with each other, rules need to be enforced in order to specify the sequence of steps to be taken. There are currently two distinct approaches to enforce the rules and carry out the interactions. They are referred to as Orchestration and Choreography. These approaches are commonly implemented when integrating multiple services in SOA. For example, they may be interacting through Web Services using SOAP messages or may be interacting using JMS messages through a messaging middleware (as described in Chapter 5 and 6).

4.2.1 Orchestration

In Orchestration, a central controller process is used to coordinate the sending and receiving of messages with other participating systems. This controller is also referred to as orchestration engine. The controller includes instructions on sending and receiving messages between multiple services. This controller may be a web service or some process instance running on the messaging middleware system. In essence, Orchestration is a way to coordinate service interaction in SOA implementations. Manolescu describes various orchestration patterns that help deal with problems when orchestration is used in enterprise architecture [20].

The main point to note about Orchestration is that all other systems don't need to know about other participating systems in the conversation. Things such as how the systems will be called, what will be the control flow, and what transformations will take place are things that are known only to the controller. The other systems only need to respond to the requests whenever they are called and do not need to maintain the state of the overall conversation. The figure below depicts how the controller coordinates the interaction among all the participating systems or services.

Orchestration provides a controlled environment that can handle different scenarios such as what to do when the response is received, when no response received, or when an error occurs. The controller needs to act according to the rules specified and the response from other systems,. For example, if an error is returned from one system, it may need to call another system, or use a default value, or compute the value to be used, or pass the control to another process.

Figure 29 - Services interaction with Orchestration [21]

4.2.2 Choreography

In Choreography, there is no central controller to coordinate the conversation with participating systems. This means every participating system needs to be aware of other systems they need to interact with. Each system needs to know which system to call next and how to communicate with that system. For example, if system A calls B, then B needs to know what to do next. It may need to respond to system A or call system C. So, the rules need to be maintained within each system instead of maintaining them within one controller process as in Orchestration [21]. It is also necessary for each system to maintain a state of the conversation or interaction in case if the conversation is transactional and needs to be rolled back upon error conditions. So, Choreography is a collaborative effort by all participating systems to utilize each others services. Orchestration, on the other hand, is a controlled and co-ordinated way of utilizing the services of all the participating systems. There is no need for each system to maintain the state of the conversation as it is maintained by the central controller. In case of error conditions, the controller knows what to do and the participating systems will be called to perform what is needed. In Choreography, however, it would require more effort to handle error conditions and maintain consistent conversational states because the state needs to be maintained inside every participating system.

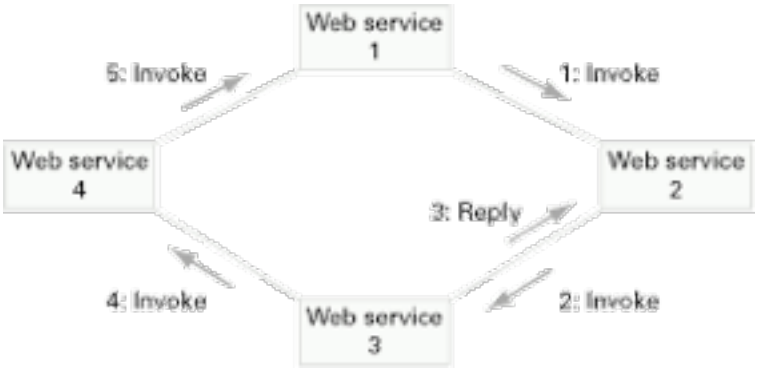

Figure 30 - Web service interaction with Choreography [21]

So far, the report has described EI from the design perspective or conceptually by analyzing different integration architectures and patterns. In the next two chapters, the focus is shifted to the implementation side by discussing the current technologies used to implement EI. In chapter 5, some of the common technologies that are currently used to implement the integration architectures and patterns will be discussed.

CHAPTER 5

MESSAGING TECHNOLOGIES

Although it is important for integration architectures and developers to know various EI patterns and architectures, knowing them is not enough. Architects and Developers need to be able to map those patterns and architectures to technology and products that suit the needs of the enterprise. So, this chapter provides an overview of some of the most current and popular technologies used in enterprise integration using messaging. It will discuss some of the open standards, protocols, languages, and platforms used to implement EI solutions based on the architectures and patterns that were discussed in previous chapters.

5.1 Extensible Markup language (XML)

XML is a markup language that is similar to HTML. The difference is that XML is designed to transport and store data with the focus on what data is, and HTML on the other hand is designed to display data with the focus on how data is displayed. XML is a simple and flexible text format derived from SGML (ISO 8879) [22] that is independent of any programming language. Using a common message format and standard like XML provides independence from programming languages and platforms. XML is currently the most common format used to construct messages. So, it enables disparate applications to communicate with each other using a common language regardless of the languages or platforms that the applications are built on. It currently plays an important role in the exchange of data in Web based applications, Web services and various Messaging middleware systems.

5.2 Web Services

Using Web Services technology is still one of the most popular ways that allows disparate applications to communicate with each other. It is built on top of open standards and platform-independent protocols. It is essentially a software interface that provides a collection of functionality or operations that can be accessed over the network through standardized messaging format such as XML [23]. So, it allows applications built on different platforms and languages to exchange messages between each other through a common interface. There are two main web services architectures to choose from, namely RPC (Remote Procedure Call family of protocols) and REST (Representational State Transfer methods). In order to choose the right standard, one needs to consider the following factors such as 1) one that is easiest to implement, 2) one that is most widely supported, and 3) one that is most likely to work well in the environment where it needs to be implemented [23]. After choosing the standard, there are many frameworks to choose from. The frameworks are essentially APIs that are

developed based on either RPC or REST based web service standards. So, the frameworks make it easier and faster to develop Web Services solution.

5.3 RPC

RPC methods allow developers to call functions similar to the way functions are used in programming single systems, but instead it calls to a remote system. Simple Object Access Protocol (SOAP) is one of the most common protocols used today that basically wraps its messages in XML and uses HTTP or JMS (Java Messaging Service) to transport the message. These functions are exposed as services through common interfaces defined by Web Service Definition Language (WSDL), and whose semantics are defined in XML. Universal Description Discovery and Integration (UDDI) is a language-independent protocol that is used to interact with registries and look for services. This makes web services discoverable to consumers. These features make SOAP based web services an appropriate choice for developing SOA applications. They are typically modeled as stateless message processors that accept request messages, process them, and typically generate a response back to the requester [23]. Although their behavior is synchronous in nature, they can be implemented as asynchronous using technologies like JMS with a messaging middleware that is discussed in section 5.5.

5.4 REST

The REST method differs from RPC because of the level at which it operates. The REST calls behave the same way as any other HTTP web requests. REST operates with stateful resources instead of individual stateless messages. So this results in a more standard and widely understood method of interacting, just like the HTTP itself. REST typically handles passing simple data blocks, whereas RPC can pass complex procedures. Although RESTful services can use SOAP as its protocol, but this is not really required because REST is simply a method of interacting, and not a protocol itself. The data can be formatted using other common language independent standards like XML or JSON (Javascript object notation). REST also does not require any additional messaging layers as SOAP does [23]. However, the notion of service discovery is not currently present in REST Web Services, unlike RPC with SOAP. Therefore, the consumer application that wishes to interact with a service provider needs an initial URI provided by the service to enter the application and consume the service.

5.5 Middleware Technology for Messaging

In order to implement messaging solutions in a scalable and loosely-coupled manner, a middleware platform is necessary to facilitate the message based interaction between multiple service providers and consumers. This type of middleware platform is also

referred to as Message Oriented Middleware (MOM). As discussed in chapter 2, a centralized broker or a message bus is needed to implement a Hub-and-Spoke or a Bus architecture respectively. In addition to providing message transport and exchange, they facilitate message routing, transformation, enrichment and so on. They are more commonly referred to as the message broker or an ESB middleware. Message broker more commonly follows the implementation of a Hub-and-Spoke architecture and ESB implementations follow the Bus architecture. But, depending on the vendor specific middleware product, there may be some overlaps between the two in terms of the functionalities and capabilities. For example, both message broker and ESB products provide message routing capabilities, but an ESB product may extend its capability by facilitating web services interaction using an orchestration engine. However, there is no concrete distinction between the two types of messaging middleware products as it is dependent on the specific vendor implementation.

In chapter 2, SOA and EDA architectures were discussed. It is important to realize that messaging middleware platforms make it possible to implement SOA and/or EDA based solutions. Web Services alone are not enough to implement these architectures because a Web Service is a point-to-point communication style by nature. For instance, it is not possible to talk to two service providers concurrently. However, a message broker or an ESB product can be setup to accept messages from any number of service providers. In turn, they can deliver a copy of each message to any number of consumers. So, Web Services interactions can become many-to-many through the use of appropriate middleware product.

5.6 Java Messaging Service (JMS)

In order for other systems to communicate with the messaging system, a messaging API is required. JMS (Java messaging service) is one of the most common messaging API and standard that is provided by many messaging middleware vendors. JMS allows application components based on the Java Enterprise Edition (JEE) platform to create, send, receive, and read messages. In an ESB based middleware, for example, JMS allows communication between different components such as router, splitter, or sequencer on the bus and enables the messages to be propagated from one component to the other. It also allows the communication between the systems to be reliable, loosely-coupled, and asynchronous. SOAP based Web Services may also be implemented over JMS to send and receive SOAP messages. This makes web services asynchronous unlike SOAP over HTTP, because JMS is an asynchronous protocol as explained next.

JMS supports two types of the message delivery models: *Point-to-Point* and *Publish-Subscribe* [24]. These two models essentially implement the message channel pattern that was described in chapter 3.

a) *Point-to-Point* - This can be used when a message needs to be delivered from a producer to one consumer. The messages are delivered to a “Queue” destination on the messaging system and then delivered to one of the consumers. For example, if a

producer system wants to send a message to a consumer system, it can send the message to a queue defined on the messaging middleware on which the consumer system would be listening on. Even if the consumer system is unavailable at the time the message is sent, the messages are held in the queue until they are consumed. So, the messages are guaranteed to be delivered to the consumer. This makes it an asynchronous communication between the producer and the consumer.

b) *Publish-Subscribe* - This model allows a producer to send messages to multiple consumers. Messages are sent to a "Topic" destination for which multiple consumer have subscribed to. So, each consumer gets a copy of the message that is sent to the this topic destination. In addition, any number of producers can send to the same topic. If there are no subscribers to the topic, the messages are generally discarded unless it has durable subscription. A durable subscription is when a consumer registered with the topic destination that can be inactive at the time the message is sent to the topic [24 20]. Alternatively, the messages on the topic may be setup to persist to a permanent storage like a database or file system using a custom application.

Each JMS message consists of three parts:

a) Header - contains information about the message itself. The information is used for routing and to determine the type of message. It has various fields as name-value pairs in which some of them are set automatically by the JMS provider when producing and delivering a message. Others are set by the client in each message.

b) *Properties* - These are optional set of values that messaging system can use to filter or route messages for example. It consists of additional information about the message such as its source and time it was created.

c) *Body* - contains the actual data for the message to be sent or received.

The following snippet of code from [24] is a sample JMS application that illustrates how a JMS message is created, sent to a Queue, and how it is retrieved from the queue. The application uses point-to-point communication to send and receive a text message. It acts as a queue sender and a queue receiver.

```
import javax.jms.*;

public class HelloMsg {
   public static void main(String argv[]) throws Exception {
      // The producer and consumer need to get a connection factory and use
it to set up
      // a connection and a session
      QueueConnectionFactory connFactory = new
com.sun.messaging.QueueConnectionFactory();
```

```
      QueueConnection conn = connFactory.createQueueConnection();
      // This session is not transacted, and it uses automatic message
acknowledgement
      QueueSession session = conn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
      Queue q = new com.sun.messaging.Queue("world");
      // Sender
      QueueSender sender = session.createSender(q);
      // Text message
      TextMessage msg = session.createTextMessage();
      msg.setText("Hello there!");
      System.out.println("Sending the message: "+msg.getText());
      sender.send(msg);
      // Receiver
      QueueReceiver receiver = session.createReceiver(q);
      conn.start();
      Message m = receiver.receive();
      if(m instanceof TextMessage) {
         TextMessage txt = (TextMessage) m;
         System.out.println("Message Received: "+txt.getText());
      }
      session.close();
      conn.close();
   }
}
```

One of the reasons that JMS is popular and commonly used is that many of the current messaging middleware products provide support for JMS. Most of these messaging middleware products nowadays follow the ESB paradigm. Next chapter will review some of these middleware products.

CHAPTER 6

MESSAGING MIDDLEWARE PRODUCTS

There are many different vendors who provide messaging middleware products, and are most commonly provided as some form of ESB implementation. Their aim is to be technology neutral, supporting multiple platforms and technologies, so that organizations can cost effectively integrate their existing systems with little or no changes. There are several commercial as well as open source products. So, choosing the appropriate middleware product is yet another challenge of EI that Architects would need to face. They would need to evaluate the products with respect to many factors such as cost, supported platforms,supported languages, performance, and so on. Happe et al present an interesting article on evaluating performance of messaging-oriented middleware [25]. The authors present model-based performance prediction methods that enable architects to evaluate the performance of MOM.

This chapter will review an open source middleware product called Mule ESB and some commercial products provided by TIBCO. The table below shows some of the most mature offerings, in alphabetical order [26]. All of them, including the open-source products, have support from one or more Vendors.

Product	Vendor	Supporting Technologies
Active Matrix BusinessWorks	TIBCO	SOAP, EMS, JMS, Rendezvous, MQ, BPEL
Mule ESB	Open-source, MuleSource, Inc.	SOAP, REST, JMS, MQ, JBI, AQ, Caching, JavaSpaces, GigaSpaces, Email, IM, JCA, AS400 Data Queues, System I/O.
OpenESB	Open-source, Sun Microsystems	JBI, JCA, JAX-RPC, JAX-WS
Sonic ESB	Progress Software	JMS, SOAP, JMX
Websphere ESB	IBM	JMS, MQ, SOAP; requires additional adapters to interface with other products and legacy protocols; requires Websphere to work

Table 1 - ESB products and their vendors[26]

6.1 Mule ESB

Mule is a java-based ESB platform that facilitates the integration of disparate applications by enabling them to exchange data between each other. It supports the integration of applications regardless of the different technologies they use such as JMS, Web Services, JDBC, HTTP, and more. At a high level, the following are the capabilities provided by Mule ESB and is based on Mule ESB version 3 as per the documentation from [27].

a) Service creation and hosting - Provides a light weight service container to expose and host reusable services

b) Service mediation - Services are shielded from message formats and protocols. Similarly, business logic is also separated from messaging protocols.

c) Message routing - Many of the message routing patterns are supported by providing content and rule based router, filter, aggregator, and re-sequencer.

d) Data Transformation - The messages can be transformed between different formats and protocols.

6.1.1 Mule ESB Messaging Architecture

There are three main layers to the architecture of Mule ESB in terms of its message flow as illustrated in Figure 31 below. The Transport Layer assists with data transmission as the message payload needs to be sent and received using different transport protocols such as HTTP and JMS. The Application Layer consists of different components or applications that implement the business logic and know how the message needs to be routed. The Integration Layer is the most important layer because it is the key to exchanging data, as it helps convert or transform the data to a format that other components can understand. It is the glue between disparate applications with different message formats.

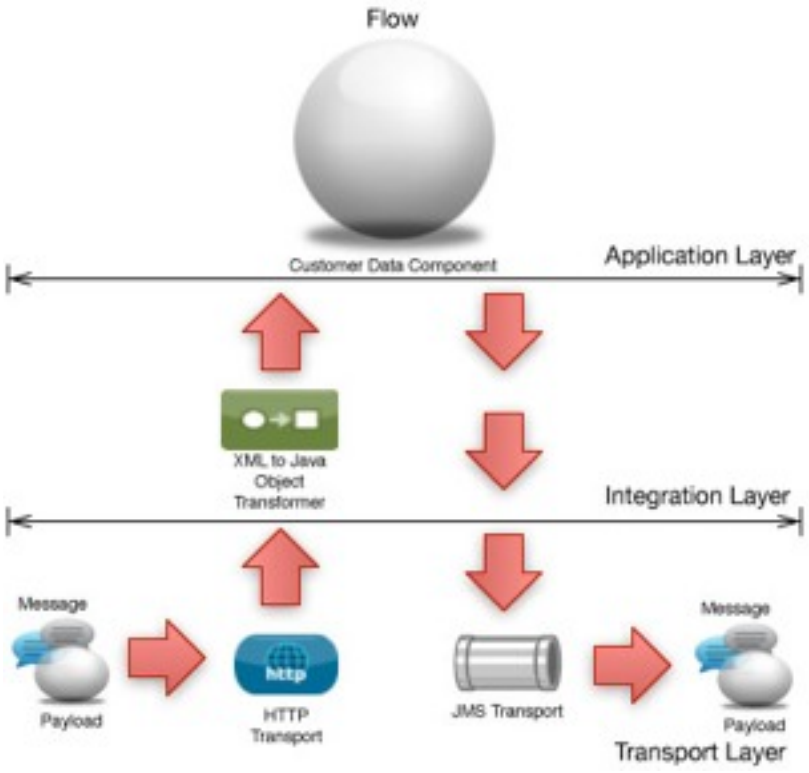

Figure 31 - Mule Architecture separated into three main layers [27]

Mule implements the following four features: Component, Mediation, Orchestration, and End-point.

a) Component - A component is implemented as a service that implements the business logic to process the message payload. When a message is sent from an application, Mule ESB picks up the message, sends it to a component that processes it using some specific business logic, and then routes it the intended application. For example, an invoice from an order entry system is sent to a component that checks the customer and inventory databases, and then it is routed to the order fulfillment system. There may be one or more components that execute business logic on messages. An example service of a component would be to read the message payload such as an order invoice and then call another component to retrieve additional client information from a customer database. Finally the component would append the information returned to the payload and then forward it to order fulfillment system. In other words, the service in this example implements the Enrichment Messaging pattern described in chapter 3. A component does not need to contain any Mule-specific code. They can be any POJOs, scripts, or web services that has the business logic for processing the data in a way that is required.

b) Mediation - The business logic implemented in the components are separated from *transport*, *transformer*, and *flow* mechanisms implemented in Mule. For example, if an XML message is sent over HTTP, firstly the HTTP *transport* mechanism turns it into a Mule message. Secondly, the *transformer* mechanism changes it along the way from XML to a Java object as required. Finally, the *flow* mechanism directs the message to each component that needs to process it. Multiple types of transports may be enabled to handle different transport protocols. For instance, a message sent synchronously over HTTP endpoint can be forwarded asynchronously over JMS after the message has been processed by a data component. This separation of business logic over transport allows for flexibility in the implementation and configuration of messaging solution.

c) Orchestration - As there may be multiple components required to process the payload, an orchestration flow wraps the components with additional message processors that transform, enhance, filter and route the messages to appropriate components. This ensures that the component receives the right messages and routes them properly after each processing. The following snippet of Mule configuration is an example of orchestration using a flow. In this flow, it accepts a book order in an XML file and transform the file's contents into a standard format that contains one or more specific orders. In the next step, the orders are split out to be processed separately, and any non-book orders are filtered out. After that, there are two services being called: one to check whether the book is in the inventory, and another to create a customer order. Both are then being e-mailed to the customer and stored in a database. If any failure occurs in this process, the message is placed in a JMS queue of failed orders and can be checked, fixed, and resubmitted later.

```
<flow>
        <file:inbound-endpoint path="/myDirectory">
                <file:filename-filter name="*.xml"/>
        </file:inbound-endpoint>
        <xml:xslt-transformer xsl-
                file="createBbookOrdersTransformation.xsl"/>
        <splitter expression="xpath://order"/>
        <!-- The following message processors will be invoked for each order in the xml
                file -->
        <expression-filter expression="xpath://
                order[@type='book']"/>
        <component class="org.my.BookInventoryChecker"/>
        <component class="org.my.BookCustomerOrderProcessor"/>
        <smtp:outbound-endpoint subject="Order Confirmation"
                        to="#{xpath://customer/email]"/>
        <jdbc:outbound-endpoint queryKey="storeOrder"/>
        <default-exception-strategy>
                <jms:outbound-endpoint queue="failedOrders"/>
        </default-exception-strategy>
</flow>
```

Code snippet: Illustration orchestration flow implemented in Mule ESB [27]

d) Endpoints - The Endpoints are configuration elements that wire together all the services. The endpoints can be specified in the flows to tell Mule ESB which transport to use, where to send messages, and which messages a component should receive. The address in the endpoint specifies the transport to use. For example, if a flow's inbound endpoint address is http://myshop.com/mule, all messages sent to that URL will be dispatched to that flow by the HTTP transport. Similarly, if the inbound endpoint is file://mymachine/files, the File transport will dispatch any new files created in that directory to that flow. Figure 32 below illustrates how endpoints play a role in the message flow.

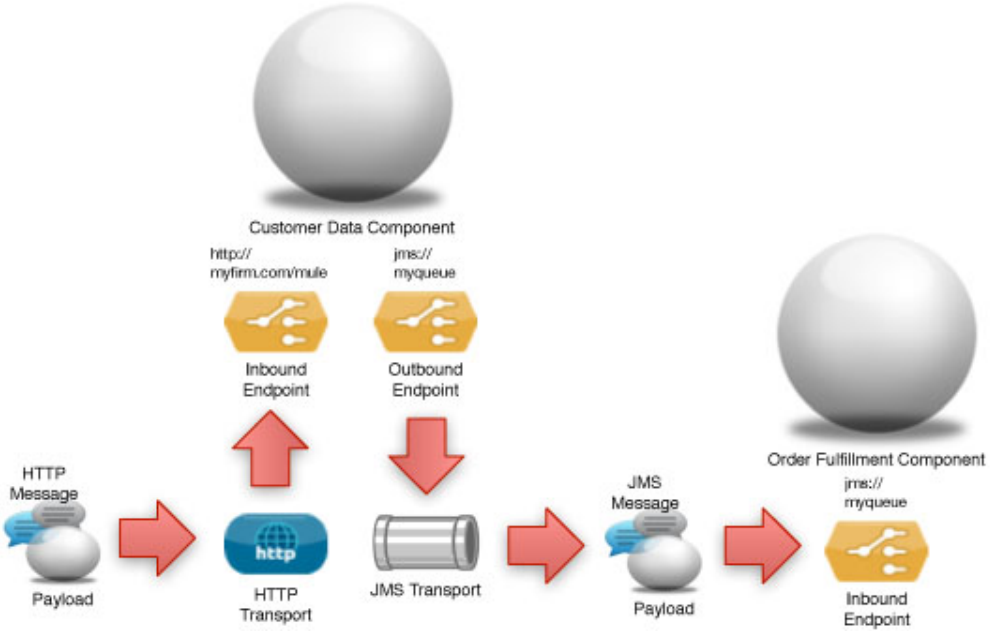

Figure 32 - Mule Endpoint configuration elements [27]

6.2 TIBCO Middleware Products

TIBCO provides several commercial EI products and tools to help enterprises design and implement their EI solutions. Their products suite is very broad and comprehensive, so it is impossible to cover all of them in this report. So, this section first provides a brief overview of some of their middleware products and development tools.

Some of the messaging middleware products offered are *Enterprise Message Service (EMS)*, *Rendezvous*, and *Message Appliance* to name a few. *EMS* is a JMS compliant messaging infrastructure that support Java, .NET, C/C++, and COBOL. It provides the foundation to build SOA [28]. *Rendezvous* is a low-latency, high-throughput messaging software for large scale distributed application environment. This is more suitable for real-time applications that require broad data distribution. The *Message Appliance* is a hardware implementation of Rendezvous. It basically replaces the standard hardware,

operating system, and messaging software with special purpose hardware to handle more message volume with less rack space and power consumption.

In addition to these middleware software and hardware products, TIBCO provides several composite applications that are technology neutral frameworks and service containers for application integration development and deployment. *ActiveMatrix Adaptors,* for instance, enable packaged applications and databases to be integrated with the messaging infrastructure and the messaging flow. So, these Adapters provide the means to easily connect existing databases and packaged applications to become participants in the enterprise information flow. *ActiveMatrix BusinessWorks* is an integration suite tool that helps to develop services and orchestrate them. It includes visual tools that enable drag-and-drop assembly of technology components and service assemblies. For example, if the application component needs to connect to a relational database, one can drag-and-drop a JDBC palette on the IDE canvas, configure the database url and credentials to make a connection to the database. So, this eliminates having to write code for such services. *ActiveMatrix Service Bus* is a lightweight ESB product that provides the means to mediate services and also includes visual tools to develop and implement the mediation between different components and services. These composite application suites and tools help enable the development of services with required business logic. These services can access existing functions across a heterogeneous environment and include them in new applications [29].

The following figure illustrates a complete product suite and how each product fits into the overall product lifecycle and integration framework.

Figure 33 - TIBCO Product suite [29]

The next sections will take a closer look at TIBCO EMS as it provides underlying message transport mechanisms and infrastructure for wide range of platforms and technologies.

6.2.1 TICBO Enterprise Message Service (EMS)

TIBCO EMS enables synchronous and asynchronous inter-system communications across a wide range of technologies such as Java EE, .NET, C, C++, and COBOL. It is a JMS compliant messaging solution that enables communication across wide range of application platforms [30]. It facilitates the implementation of various routing and interaction patterns by configuring Queues, Topics and Bridges for the desired message flow. Some of the patterns supported are point-to-point, publish-subscribe, and request-reply. Chapter 5 already described JMS Queue and Topic. A Queue is used to implement point-to-point routing of message. A Topic is used to implement publish-subscribe pattern to route messages to multiple targets. However, there are different ways to configure the routing.

a) *Point-to-Point routing in EMS*

In EMS, the concept of a "Bridge" is used to route messages between Queues and Topics [31]. For example, a message that is published to a Topic can be bridged to a Queue. This Queue may be a dedicated input Queue for the target system to receive the messages. If messages published to the Topic need to be persisted to a database, for example, a component may be implemented to persist any messages published to that Topic. This is one example on how to implement point-to-point routing in EMS as illustrated in Figure 34 below.

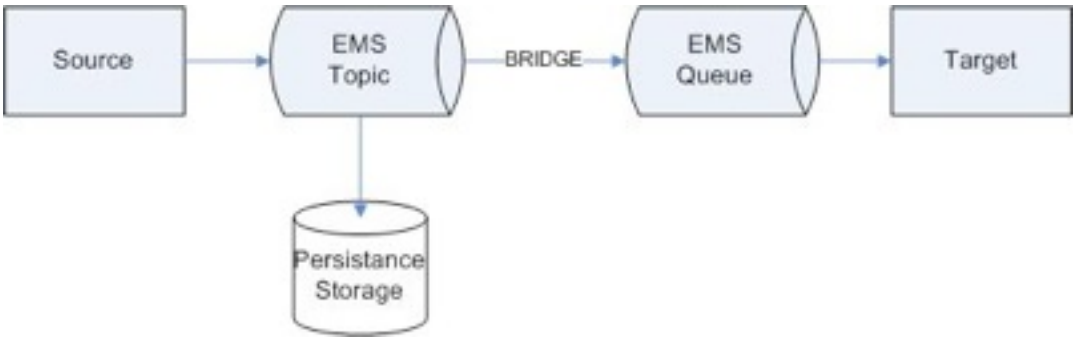

Figure 34 - Point-to-Point routing in EMS

Publish-Subscribe routing in EMS

If a message needs to be routed to multiple Queues from a Topic, then multiple bridges can be created from that Topic to route messages to multiple Queues and thus implementing a publish-subscribe pattern. Each bridge created can be configured to route only certain types of messages based on a selected criteria. It is also possible to subscribe to the Topic instead of having to bridge the Topic to a Queue. However, if the target system is not available at any given time then message would be discarded

without being consumed by the target system. So, routing the message to a Queue would guarantee delivery. The following Figure 35 illustrates this scenario.

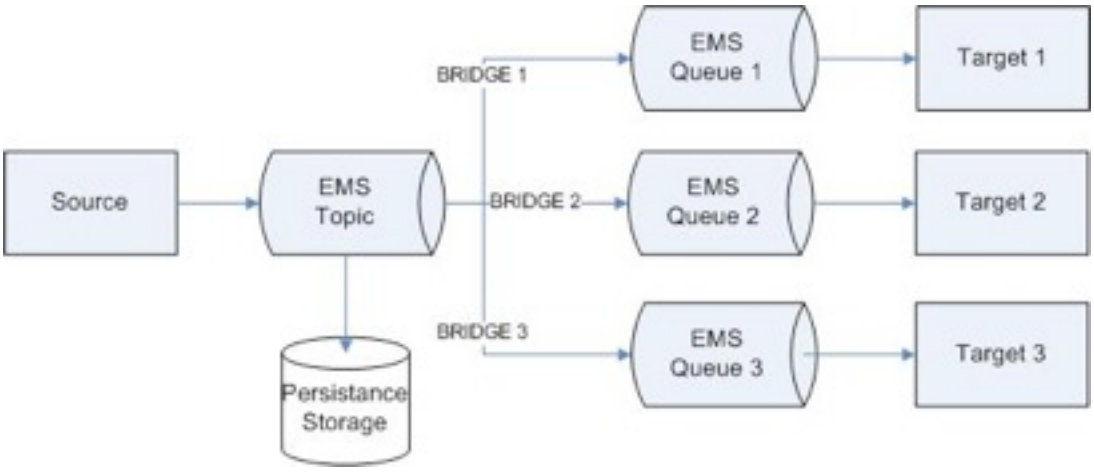

Figure 35 - Publish-Subscribe routing in EMS

c) *Request-Reply interaction pattern in EMS:*

One way to implement a Request-Reply pattern would be as follows. The source system can publish to a Topic that is bridged to a Queue for the target system to receive the message. The target system may then reply by publishing to a specific reply Topic that is bridged to a Queue for the Producer to retrieve the response message. If multiple target systems need to reply to the same message, then all of them can publish their response messages to the same reply Topic. Figure 36 below illustrates this scenario.

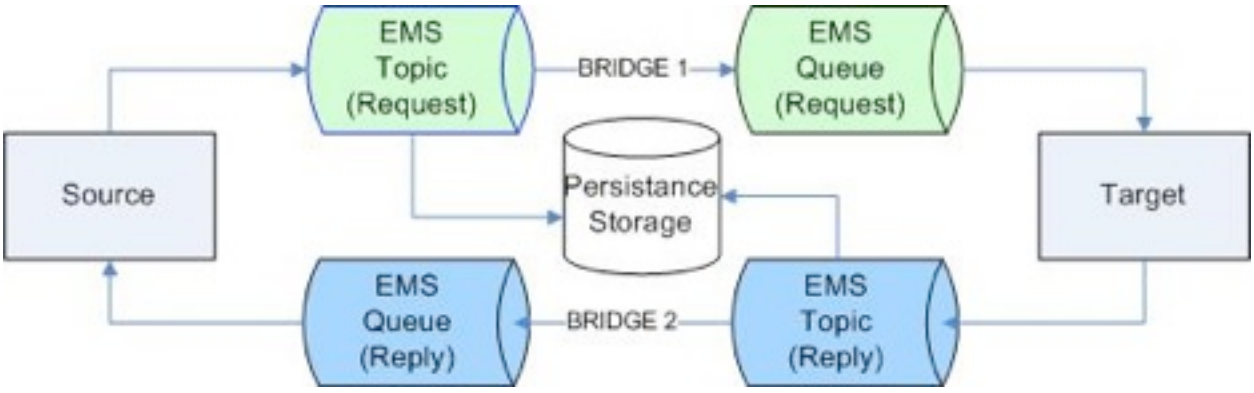

Figure 36 - Request-Reply interaction pattern in EMS

CHAPTER 7
CONCLUSIONS AND RECOMMENDATIONS

7.1 Conclusions

The main objective of this research was to illustrate how disparate enterprise applications can be integrated using Messaging. It first provided an overview of what enterprise integration is and why it is important for business organizations. It described why messaging is a more suitable solution compared to other styles of EI. Various integration architectures and patterns were also discussed on how they help solve many common integration problems. In addition to the architecture and design side of EI, the essay has described many of the current integration technologies as well as products that organizations can use to implement their EI solutions. So, this report has covered the subject from the design perspective as well as implementation perspective.

One of the most important things to realize, when implementing EI, is that it is not feasible for organizations to completely change their existing systems and infrastructure in order to integrate their systems. Also, organizations don't always implement their I/T infrastructure and systems from the ground up knowing how they are going to be integrated. As the business evolves and changes quickly, the systems built to support it have to change and adapt quickly as well. Some of these systems may have been built in isolation and never intended to work with each other. They may have been built using different programming languages and platforms. So, it is also important for integration solutions to be technology neutral. This means the EI solution has to be able to support multiple existing platforms, network protocols, languages that the existing systems were built around. It is because of these reasons, integrating those disparate systems using appropriate messaging architecture and middleware products can be a feasible solution.

The ESB model fits these requirements very well as it facilitates the implementation of architectures such as SOA and EDA. SOA leverages existing systems by wrapping them and exposing their functionality as discoverable services. However, this is only the first step towards enterprise integration. Once the functionalities are provided through a common interface, the data sharing and interaction with other services need to be enabled in a decoupled and a scalable manner. This is where the importance of implementing the appropriate interaction patterns come into play. For instance, the message exchange may need to be synchronous or asynchronous. It may need request-reply or request-reply-retry interaction enabled. The appropriate interaction patterns would need to be implemented. In addition, multiple interacting services may need to be implemented by following the appropriate models such as choreography or orchestration that enforce the rules around how different services need to work with each other.

In business systems, different business events may trigger business processes . An example of an event would be when a company requests a quote from multiple suppliers. When each supplier system receives the request for a quote, they may need to interact with other internal systems or services before a response can be sent back to the requestor. When the requestor receives all the responses, it then needs to evaluate all the responses from different supplier systems and make a decision. This means implementing EDA with SOA would be suitable as requests and responses (events) need to be fired and propagated to invoke and interact with various services in real time. In order to facilitate the interaction between these services, a messaging middleware product such as TIBCO EMS with BusinessWorks container or IBM Websphere ESB would be two examples of suitable middleware products for such business scenario. A messaging engine and components deployed on the messaging infrastructure can propagate the messages between different services. Operations such as validate, transform, route, enrich, and filter would need to be performed to the messages as necessary. If different enterprises use different messaging middleware, then it may be necessary to use a common supporting messaging standard such as JMS. This allows two different vendor products to be integrated using messaging. Therefore, the knowledge of various messaging patterns, messaging middleware products and open standards is necessary to implement the appropriate integration solution. This knowledge can help solve many common integration problems such as routing between different transport protocols, message formats, and platforms.

With respect to implementing technologies, however, challenges can still occur even if the design is sound. For example, limitations with the technology or products may arise in different areas like security, latency, throughput, storage, and so on. Most of these issues may be discovered during the analysis phase and by performing proof of concept for the design using the chosen technology and products. But, some issues may not be detected until the system implementation is in production for some time. So, continued changes and fixes will still be required in an integrated environment as with any systems.

Troubleshooting and recovering from problems that occur in an integrated environment with multiple systems would also be much more challenging than when they occur in a single system. Following good design practices such as integrating them in a decoupled and asynchronous manner can ease those challenges. For instance, if systems are tightly coupled with synchronous communication, it requires all systems to be active and ready to process at all times in a given workflow of activities. If one system goes offline, the entire workflow would be affected. In this scenario, there may be no easy way to recover from the crash other than having to restart the process from the beginning. However, if the communication is asynchronous in nature, the workflow may be re-started from where it was left off, when the inactive system becomes active again.

Similarly, the maintenance of tightly coupled systems is also more difficult to manage. If one system changes its interface, it would require all other systems to be updated as well, because they explicitly need to know how to communicate with that system. If a

messaging middleware is put in place, for instance, it can handle the routing and communication with other participating systems. Each system only needs to know how to communicate with the messaging middleware and not each other. Any changes would only have to be done to the system that requires the change, and the adapter or endpoint that implements the communication with the messaging system.

Although messaging has many advantages and provides many benefits when implementing EI, it may not always be the right choice for all business scenarios. One of the primary disadvantages is that it requires an additional component in the architecture such as the messaging system itself. In some cases, this may even cause reduction in performance and reliability. For instance, it may not be suitable for small organizations. Instead, it may be simpler to have dedicated point-point to connections if they only have a few systems. Another example would be when systems require large files to be transferred from one to the other. In this case, messaging may not be suitable because it would cost more for the organization to implement a messaging solution as it would require significant amount of changes to the way it handles its data currently. Messaging solutions would also require the organizations to provide additional training to existing staff or even hire new staff with the appropriate skill sets to help support the new integrated environment. Implementing SOA for existing legacy systems is yet another example of additional cost for the changes required. Although SOA can provide legacy enablement and help integrate those systems with modern technology and expose them through different application interfaces, it has its own challenges to overcome as well. For example, financial institutions would need to ensure that their existing system functions exposed as services are still secure and robust. They need to ensure that their network and data are not compromised by the new messaging protocols and products used to integrate with other systems.

7.2 Recommendations

Overall, this essay dwells into two major challenges in current EI practice. The first challenge is to choose the appropriate architectural design and proven integration patterns based on the organization's business requirements. The second challenge is to be able to map that design to the appropriate technology and products that are suitable for the organization and its current I/T infrastructure. This report provides the necessary foundation to start tackling these challenges from a technical perspective. But, further research is necessary on how to map the organization's business requirements to technical requirements as it is important for the integration developers to realize the need for EI for the business. Therefore, it is necessary to ask the question of "why" EI is required and not just "how" EI can be implemented. This essay helps to answer the question of "how" by providing the knowledge on proven architectures, patterns, as well as technology that can help solve commonly occurring technical problems in EI.

REFERENCES

[1] Anaya, V., Ortiz, A., (2005), “How Enterprise Architectures Can Support Integration”, Copyright © 2005 ACM

[2] Mengjian, Chen., (2009), “Research and Implementation on Enterprise Application Integration Platform”, IEEE Xplore

[3] Gorton, Ian, Liu, Anna., (2004), “Architectures and Technologies for Enterprise Application Integration”, IEEE Computer Society

[4] Noble, James. (2007). “Every Good Designer Uses Patterns”. Copyright © 2007 IEEE Software

[5] Gamma, E; Helm, R. Johnson, R. Vlissides, J., (1995). “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley.

[6] Martin, F. (2002). “Patterns of Enterprise Application Architecture”. Addison-Wesley.

[7] Hohpe, Gregor., Woolf, Bobby., (2003), “ Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”, Addison-Wesley

[8] Goel, Anurag., " Enterprise Integration - EAI vs. SOA vs. ESB", Retrieved on March 20th, 2011 from http://ggatz.com/images/Enterprise_20Integration_20_20SOA_20vs_20EAI_20vs_20ESB.pdf

[9] Lee, J., Siau, K., Hong, S., (2003), “Enterprise Integration with ERP and EAI”, Copyright © 2008 ACM

[10] Chatterjee, Soumen., (2004), “Messaging Patterns in Service-Oriented Architecture”, Retrieved on September 2011 from http://msdn.microsoft.com/en-us/library/aa480027.aspx

[11] Ghalsasi, Sadhana Yogesh. (2009), “Critical Success Factors for Event Driven Service Oriented Architecture“, Copyright © 2009 ACM

[12] Michelson, Brenda M., (2006), “Event-Driven Architecture Overview - Event-Driven SOA Is Just Part of the EDA Story”, Patricia Seybold Group

[13] Chappell, Dave., (2004), "ESB Integration Patterns", Retrieved on April 1st, 2011, from http://soa.sys-con.com/node/46170

[14] Lublinsky, Boris, (2009), “An ESB Vendors Evaluation by Forrester Research”, Retrieved on March 22nd from http://www.infoq.com/news/2009/02/ESBVendors

[15] W3C, (2003), “Web Services Architecture”, Retrieved on August 2003 from http://www.w3.org/TR/2003/WD-ws-arch-20030808/

[16] Holpe, Gregor (2003), “Enterprise Integration Patterns - Asynchronous Messaging Architectures in Practice”, JAOO Conference - Copyright © 2003 Gregor Holpe

[17] Holpe, Gregor, (2011), “ Enterprise Integration Patterns”, Retrieved on April 7th, 2011 from http://www.eaipatterns.com/

[18] Holpe, Gregor., (2007), “Conversations between loosely coupled services”, Retrieved July 2011 from http://www.infoq.com/presentations/hohpe-soa-conversations

[19] Barros, A., Dumas, M., Hofstede, A.T, (2005), “Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection”, Copyright © 2005 The Authors, Faculty of IT, Queensland University of Technology

[20] Manolescu, Dragos A., (2004), “Patterns for Orchestration Environments”, Copyright © 2004 Dragos A. Manolescu, hillside.net

[21 19] Geek Explains, (2008), “Ways of combining Web Services, Orchestrations vs Choreography”, Retrieved on August 15th, 2011 from http://geekexplains.blogspot.com/2008/07/ways-of-combining-web-services.html

[22] W3C, (2011), “Extensible Markup Language”, Retrieved on October 19th, 2011 from http://www.w3.org/XML/

[23] Schluting, Charlie., (2009), “Understanding SOA Technologies”, Retrieved on April 5th, 2011, from http://www.enterprisenetworkingplanet.com/netsp/article.php/3806171/Understanding-SOA-Technologies.htm

[24] Oracle (Sun Developer Network), (2004), “Getting Started with Java Message Service”, Retrieved on November 06th, 2011, from http://java.sun.com/developer/technicalArticles/Ecommerce/jms/

[25] Happe, J., Friedrich, H., Becker, S., Reussner, R.H., (2008), “A Pattern-Based Performance Completion for Message-Oriented Middleware”, Copyright © 2008 ACM

[26] TheServerSide.com, (2007), “Mule: A Case Study”, Retrieved on November 7th, 2011 from http://www.theserverside.com/news/1365047/Mule-A-Case-Study

[27] MuleSoft, (2011), “Mule ESB 3 concepts”, Retrieved on November 10th, 2011 from http://www.mulesoft.org/sites/all/themes/litejazz/docs/mule-esb-3-concepts.pdf

[28] TIBCO, (2011), “Messaging”, Retrieved on November 18th, 2011 from http://www.tibco.com/products/soa/messaging/default.jsp

[29] TIBCO, (2011), “Building Composite Applications”, Retrieved on November 20th, 2011 from http://www.tibco.com/products/soa/composite-applications/default.jsp

[30] TIBCO, (2009), “TIBCO Enterprise Message Service”, Retrieved on November 18th, 2011 from http://www.tibco.com/multimedia/ds-message-service_tcm8-824.pdf

[31] Lawrence, Richard., (2004), “TIBCO EMS Routing Design Patterns”, Retrieved on November 29th, 2011 from https://ssl.tibcommunity.com/servlet/JiveServlet/previewBody/2597-102-1-2503/EMS-Routing-Design-Patterns-August2007.pdf

[32] Robinson, Rick, (2004), “ Enterprise Service Bus Patterns”, IBM Software Group, IBM Corporation

[33] Hoof, Jack van., (2006), “How EDA extends SOA and why it is important”, Retrieved on October 9th, 2011 from http://soa-eda.blogspot.com/2006/11/how-eda-extends-soa-andwhy-it-is.htm

[34] Hohpe, Gregor., (2007) “SOA Patterns – New Insights or Recycled Knowledge?”, Google Inc.

[35] Linthicum, David S., (2000), “ Enterprise application integration”, Addison-Wesley Information Technology Series