Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis No. 3480

Complete Enterprise Topologies with routing information of Enterprise Services Buses to enable Cloud-migration

Andre Grund

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Tobias Binz
Commenced:	May 01, 2013
Completed:	October 28, 2013
CR-Classification:	E.1, K.6

Abstract

The Enterprise Service Bus is an important part of todays enterprise IT landscape. It offers the integration of applications build on different platforms without adaptation. This is accomplished by offering message transformation and routing capabilities of client requests to the designated endpoint service. However, Enterprise Service Buses also introduce an additional indirection between the client and the called backend application.
Enterprise Topology Graphs capture a snapshot of the whole enterprise IT and are used in various use cases for analysis, migration, adaptation, and optimization of IT. The focus of this work is to enhance the ETG model with structural and statistical information about an enterprise. However, due to the decoupled architecture the information is hidden inside the ESB and not directly accessible. Furthermore, the arrangement and semantics of the routing entities are unknown.
The existing ETG Framework includes the automated discovery and maintenance of ETGs, but offers no solution for ESB components in the enterprise IT. This thesis provides an in depth analysis of the ESBs Apache Camel and Apache Synapse. It applies information gathering concepts and evaluate them with a prototypical implementation of an ETG Framework plugin. Using tailored information gathering and presentation methods to enhance ETGs with routing information. The result has been evaluated using scenarios provided by the ESBs, including a detailed actual-target comparison.
With this thesis, fundamental concepts for routing information gathering from ESB have been developed. Thereby, the routing information and statistics are gathered into a generic data model which has been defined to universally model ESB information of different ESBs. In the last step, this information is used to complete and enhance the ETG with ESB routing information and statistics. This work closes a gap in the ETG coverage and completes the ETG by providing insight into the relations of the different enterprise IT components.

Contents

1 Introduction 11
- 1.1 Motivating Example 11
- 1.2 Problem Statement 12
- 1.3 Outline 13

2 Fundamentals and State of the Art 15
- 2.1 Cloud Computing 15
- 2.2 Enterprise Service Bus 16
- 2.3 Enterprise Topology Graphs 17
- 2.4 ETG Framework 18
- 2.5 Apache Camel 19
- 2.6 Apache Synapse 20
- 2.7 Information Gathering Strategies 21

3 Focus on Apache Camel 23
- 3.1 Classification 23
- 3.2 Architecture and Concept 24
- 3.3 Pattern Overview and Structure 27
  - 3.3.1 Splitter 28
  - 3.3.2 Recipient List 29
  - 3.3.3 Pipes and Filters 30
  - 3.3.4 Aggregator 31
  - 3.3.5 Message Router 32
  - 3.3.6 Dynamic Router 33
- 3.4 User Stories 33
  - 3.4.1 Apache ServiceMix 33
  - 3.4.2 Fuse ESB Enterprise 34
- 3.5 Camel Technologies 35
  - 3.5.1 Apache CFX 35
  - 3.5.2 Direct Component 35
  - 3.5.3 Jetty Component 35
  - 3.5.4 SEDA Component 36
  - 3.5.5 Spring Framework 36

3.6 Evaluation Scenarios . . . 37
3.6.1 Load Balancer . . . 37
3.6.2 Business Example - The Coffee Shop . . . 39
3.7 Applied Gathering Methods . . . 43
3.8 Crawler Implementation Details . . . 45
3.8.1 Crawler Data Model . . . 45
3.8.2 JMX Object Helper . . . 46
3.8.3 Camel Structure Builder . . . 47
3.8.4 Camel JMX Connection Client . . . 48
3.9 Discussion of Crawling Results . . . 49

4 Focus on Apache Synapse 51
4.1 Classification . . . 51
4.2 Architecture and Concept . . . 52
4.3 Functional Components . . . 54
4.3.1 Sequence . . . 54
4.3.2 Proxy . . . 55
4.3.3 Endpoint . . . 56
4.3.4 Mediator . . . 57
4.3.5 Core Mediators . . . 57
4.3.6 Filter Mediators . . . 58
4.3.7 Transformation Mediators . . . 59
4.3.8 Extension Mediators . . . 60
4.3.9 Advanced Mediators . . . 61
4.4 Pattern Overview . . . 63
4.5 User Stories . . . 65
4.6 Evaluation Scenarios . . . 65
4.6.1 Content Based Routing Example . . . 66
4.6.2 Aggregation and Splitter Example . . . 67
4.6.3 Load Balance Example . . . 69
4.7 Information Gathering Methods . . . 70
4.8 Crawler Implementation Details . . . 71
4.8.1 JAXB parsing . . . 71
4.8.2 Crawling Static Routing Information . . . 72
4.8.3 Crawling Routing Statistics . . . 75
4.9 Discussion of Crawling Results . . . 75

5 Generic Model for Enterprise Service Bus Routing 79
5.1 Usage and Requirements . . . 79
5.2 Model Specification . . . 80
5.2.1 Model Structure . . . 80
5.2.2 Entity Descriptions . . . 80
5.2.3 Entity Characteristics . . . 81
5.2.4 Best Practices . . . 82
5.3 Interpretation and Design . . . 84

5.3.1 Data Model 84
5.4 ETG Builder 85

6 ETG Framework Integration 87
6.1 Enterprise Service Bus Discovery 87
6.2 ESB Structure Rendering in the ETG 88
6.3 Apache Camel Plugin 90
6.4 Apache Synapse Plugin 90
6.5 Evaluation 91

7 Summary and Outlook 93

A Synapse Deployment and Installation Guide 95
A.1 Deploying Apache Synapse 95
A.1.1 Pre-requisites 95
A.2 Step by Step Guidance 96
A.3 LoadBalancer Example 96

Bibliography 97

List of Figures

3.1 Illustration of the main components of Apache Camel 26
3.2 Structure of the load balance example evaluation 39
3.3 Structure of the expected splitter route elements 42
3.4 The recipient list with the unknown target endpoint. 42
3.5 Preparation routes of either cold or hot drinks. 43
3.6 The expected aggregator route elements. 43
3.7 The camel crawler data model depicted as UML class diagram 46
3.8 The Apache Camel internal crawler structure 47
3.9 Example of several interpretation alternatives of two Apache Camel routes . . 49

4.1 Overview of the main components of the Apache Synapse architecture 53
4.2 The Synapse messaging flow structure 55
4.3 Illustration of the create packages from JAXB xjc 72
4.4 The definition of the message flow elements proxy and sequence 73
4.5 Transformation example of the Apache Synapse model to the generic ESB data model 76

5.1 Base call of all generic data model entities 84
5.2 Illustration of the processing steps of the ETG route builder. 85

6.1 Synapse and Camel application topology rendered without routes as Enterprise Topology Graph. 87
6.2 Example of an complete Apache Synapse topology with two routes illustrated as ETG 89

List of Tables

3.1 Comparison of ESB characteristics and Apache Camel functionality 24
3.2 Apache Camel splitter 28
3.3 Apache Camel recipient list 29
3.4 Apache Camel pipes and filters 30
3.5 Apache Camel aggregator 31

3.6 Apache Camel message router 32
3.7 Apache Camel dynamic router 33

4.1 Comparison of ESB characteristics and Apache Synapse functionality 52
4.2 Correlation of Synapse mediators and Enterprise Integration Patterns 64

5.1 Characteristics of the generic model entities 81
5.2 Mapping between Enterprise Integration Patterns and Generic Data Model elements 83

List of Listings

3.1 The syntax of the Apache Camel JMX output 27
3.2 Service interface of the Calculator WSDL file 37
3.3 Example of one endpoint configured as jaxws endpoint 38
3.4 Calculator example route using a load balancer with four endpoints 38
3.5 First coffee shop route from *direct:cafe* invoking a splitter then send to *direct:drink* 40
3.6 Second coffee shop route from *direct:drink* invoking a *recipient list* bean 40
3.7 Coffee shop preparation route alternative A for cold drinks 41
3.8 Final route aggregating the orders, preparing and finally deliver the drinks . . 41
3.9 Example of an camel route configuration using Java DSL 44
3.10 MObjectHelper: Query an endpoint using the *getObjectInstance* method 46
3.11 MObjectHelper: Query a arbitrarily object instance in JMX 47
3.12 Simplified Apache Camel structure clustering algorithm 48

4.1 The syntax outline of a sequence definition 54
4.2 Example of an main Synapse sequence with three mediators 54
4.3 The syntax outline of an Apache Synapse proxy service 56
4.4 Example: Content based routing with Apache Synapse 66
4.5 Example: Aggregation and Splitter with Apache Synapse 67
4.6 Example: LoadBalancer with Apache Synapse 69
4.7 Starting the RouteBuilder with proxy and routes 73
4.8 Route Builder algorithm building the AbstractRoute 74
4.9 Information Assembler 74

6.1 HTTP Response header of an HTTP Request 88

CHAPTER 1

INTRODUCTION

This Chapter introduces the context and motivation of this diploma thesis. It discusses the need of enhancing enterprise typology with routing information from Enterprise Service Buses. This will be illustrated with an example that shows the need to solve the unknown routing problem. In addition the problem statement and the outline is given.

1.1 Motivating Example

Today enterprises have to be competitive in an global market that challenge a business to reduce costs. Additional in times with slowdown in the global economy the pressure on the IT spending will rise [Roe12].
Service-oriented computing offers a flexible infrastructure and processing environment for reusable application functions as services. The services reflect a "service-oriented" approach that is based on the idea of composing applications. Web services make use of open standards, that enable easily integration of application components into a loosely coupled network of services. The requirements of SOA and Web services can be adapted to the concept of the Enterprise Service Bus [PTDL07]. Large international enterprises like Hermes [Red12] and Deutsche Post [ES08] already operate on a heavily SOA orientated structure with an sophisticated ESB product. Additionally, the worldwide SOA market will grow steadily and should strengthen the ESB as the dominant application-integration approach [Ort07]. Through the fast changing IT environment administrators often loose track of their infrastructure and struggle to discover dependencies between different parts of their components and their orchestration [BBKL13]. The rising complexity and the huge amount of components makes a manual modeling of the IT structure a time-consuming, costly, and error-prone task.
The ETG Framework enables the automated discovery of components, business processes, software and infrastructure. However, today the ESB information are not represented in the Enterprise Topology Graph. The reason is that several challenges remain unsolved regarding the discovery and analysis of ESBs. The internal message routing and Web services are not visible from the outside. More precisely, the ESB places an indirection between the requesting client an the called service. This is also called the *unknown routing problem*, because the ESB conceals the message flow and the identity of the back-end application. These predefined

routes are defined by the static configuration of the ESB.
In addition, besides the static route information the IT administration are also interested in statistical data about the internal message routing and utilization of applications. Typically a request invokes a linear route that handles the message directly to an endpoint. But its also possible building routes with dynamic elements that effects the routing outcome. They are called routing mediators and can implement arbitrary routing strategies. For instance, the ESB could determine an endpoint based on the message content or on the endpoint workload. The ETG Framework facilitates specialized plugins that discovers different components and adds them to the ETG [BBKL13]. The Enterprise Topology Graph describes an enterprise topology as an formal model based on established graph theory [BFL$^+$12]. Features like structural queries and sophisticated search algorithm can be applied for IT consolidation to reduce operational costs. Additional plugins are needed with ESB specific concepts to gather static routing information and statistical data. Both enterprise decision makers and IT administrators would benefit from automated discovery and documentation including state of the art enterprise middleware. ETGs enhanced with detailed ESB information could lead to further consolidation, migration, or outsourcing of IT.

1.2 Problem Statement

This work implements concepts to extract static routing information and statistical results from the ESB products Apache Camel and Apache Synapse. This research is evaluated with a plugin implementation for each respective ESB in the ETG Framework.

- **Research general concepts of Apache Synapse and Apache Camel**
  The products are different in their architecture and used technologies to implement an Enterprise Service Bus. This includes the message mediators and their orchestration as ESB routes. The domain specific semantic has to be correctly represented in the ETG.

- **Concepts to solve the unknown routing problem**
  The ESB is designed to decouple the client request from the back end application. This indirection is called the unknown routing problem. These hidden information have to be made accessible.

- **Concept to represent the information in the ETG**
  The ETG model can describe enterprise topology including every component of the IT infrastructure. The Enterprise Service Bus implements various routes, mediators and endpoints. They have to be integrated in the structure of the ETG graph. Finally this task needs to adapt the ETG model, while providing semantic correctness and easy adaptation on different ESB products.

- **ESB information crawler implementation**
  Apache Camel and Apache Synapse provide different implementation of an ESB that needs custom solutions to gather routing information. Moreover, these concepts should provide minimal operational impact for the enterprise that utilizes the ETG Framework.

So it is elementary to choose a suitable data crawling method. This concept needs to be implemented and provided as a tested and evaluated crawling application.

- **Evaluation Scenarios**
  A creation of suitable test scenarios is needed for the research, implementation and the final evaluation of the implemented ESB plugin. Besides the examples should cover the main features and patterns of the ESB.
- **ETG Framework Integration**
  The ESB crawling concepts have to be integrated in the ETG Framework. It is also needed to discover running Apache Camel or Apache Synapse instance. The crawling result have to be imported to the ETG model.
- **Generic model for Enterprise Service Bus routing rules**
  After all the generic model should assist ETG Framework plugin developer to provide routing information in the ETG model. It unifies the representation with defined semantics and correlation of ESB entities.

1.3 Outline

This document describes the fundamentals and concepts to complete enterprise topologies with routing information.
Foremost, the necessary fundamentals of this research will be discussed. It covers the cloud computing paradigm in Section 2.1, followed by the Enterprise Service Bus technology in Section 2.2. Section 2.3 and 2.4 discusses the ETG and the corresponding Framework. Additionally, the ESB products Apache Camel in Section 2.5 and Apache Synapse in Section 2.6 will be briefly summarized. Finally, the available information gathering methods are discussed in Section 2.7.
Chapter 3 and 4 focus on the Enterprise Service Buses Apache Camel and Apache Synapse. Initially, both applications will be classified according the ESB characteristics formed by D.A. Chappell. Hereafter the ESB Architecture and configuration entities will be discussed. Both Chapters review user stories that already make use of the specific product. Finally the developed crawler application will be presented. Beginning with the evaluation scenarios followed by an in depth implementation description.
During the implementation the possible benefits of an generic data model for enterprise service buses became obvious became necessary. This model for ESB routes information will be part of the 5th Chapter.
The findings and implementation of the previous Sections will be integrated in the ETG Framework Chapter 6.
Eventually, the work of this document will be wrapped up in a summary and a outlook will be given.

CHAPTER 2

FUNDAMENTALS AND STATE OF THE ART

In this Chapter the fundamental and state of the art concepts and technologies are discussed. These sections are providing the needed knowledge to understand the key concepts connected with this thesis. The first section 2.1 explains cloud computing and the significant in todays enterprise IT. The Enterprise Service Bus basics and characteristics are discussed in 2.2. Furthermore the enterprise topology in section 2.3 with the ETG Framework covered in 2.4 can be used to formalized and automate discovery of IT architecture.
Finally the ESBs Apache Camel in Section 2.5 and Apache Synapse in Section 2.6 will be introduced. Followed by the fundamental information gathering strategies in Section 2.7.

2.1 Cloud Computing

Cloud computing is a new paradigm and has recently attracted significant momentum which was mainly driven by the industry. Jeffrey Voas and Jia Zhang call cloud computing a new phase in the computing paradigm shift of the last half century [VZ09]. IT technology has begun with terminals connected to centralized mainframe computers. Then the decentralized computers became powerful enough to satisfy users daily work. Finally network technology connected all the standalone computers together and utilized remote applications and storage resources. In todays IT evolution all resources will be exploited and made available over the Internet. This technology can be called cloud computing.
Yet there exists no commonly accepted definition for cloud computing [Erd09].
The NIST[1] definition of cloud computing describes the technology with several essential characteristics [MG09]:

- **On-demand self-service**
  The customer can acquire computing capabilities automatically on request. An analogy

[1]National Institute of Standards and Technology - http://www.nist.gov/

of this is the power grid distribution system that offers customers power without requiring human interaction as customer service from the power supplier.

- **Broad network access**
  The service is available over the Internet through standard mechanism.
- **Resource pooling**
  All resources are pooled to serve multiple customers. The computing capacity is shared without exposing the exact location of the resource.
- **Rapid elasticity**
  The service can be purchased and used in any demanded quantity at any time. The underlying capabilities are automatically provisioned in a rapidly and elastically way.
- **Measured service**
  The service can be automatically controlled by a metering capability that is appropriate to the kind of service.

The NIST definition also groups service models that can be offered through cloud computing. The core criteria focus on cloud architecture, virtualization and services. The service is arranged in defined "XaaS" where X represents what the service basically is offering, for instance SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service) or IaaS (Infrastructure-as-a-Service) [RCL09]. There are key advantages and opportunities that cloud computing can offer. The technology can dramatically lower the costs for the IT infrastructure of a business. It is possible to acquire a large amount of computer power for relatively short amounts of time and at low costs. Furthermore the upfront capital investments will be lower when a business applies to a cloud computing product. Also, an enterprise can easily scale their computing capacity based on the demand of the services. These advantages drive the development of this new paradigm in industry and research [MLB+11].

2.2 Enterprise Service Bus

The Enterprise Service Bus is a result of an evolutionary process of existing integration architectures. The Message Orientated Middleware was created in the early 90's and was the first approach in the field of product integration. The technical advances lead to application servers and Web services. The common goal of these technologies is the integration of enterprise applications. The Enterprise Service Bus represents a new infrastructure as a combination of existing technologies. These components are among others message-oriented middleware, web services, transformation and routing intelligence. But these are just the ingredients of an Enterprise Service Bus and not a reliable definition or architecture. The technology is driven by the needs of the industry and built to solve problems and reduce IT costs. So there is a market with many competitors trying to gain attention in this growing IT business. In fact many products claiming to be an Enterprise Service are leading to a blurred meaning and confusion what an Enterprise Service Bus really is. Basically David Chappell is using the following characteristics as a reference:

- **Pervasiveness**
  The ESB should be the basis of all applications in the enterprise network. New applications can be plugged and unplugged to the system and can all be reached through the bus. It is a unified messaging system that decouples the applications and makes them available in an entire enterprise network.
- **Standards-Based Integration**
  It should be easy to integrate applications based on a variety of technologies. The ESB should provide adapters to integrate an application to the Enterprise Service Bus. Standards like SOAP, XML, WSDL or BPEL4WS are supported out of the box. Also adapters can be used to connect applications built with .NET, COM, C#, J2EE and C/C++ without changing their implementation.
- **Highly distributed, event-driven SOA**
  There is no hub and spoke architecture with a tightly coupled server. All components on the bus can be independently integrated and are globally asynchronous accessible on the bus.
- **Security and reliability**
  The transport between nodes in the ESB has to be firewall-capable. Authorization, credential-management and access control has to be ensured by the bus. The ESB has to provide asynchronous communication and reliable delivery of business data with transaction integrity.
- **Orchestration and process flow and data transformation**
  The ESB must orchestrate processes as messages flow ranged of any complexity. The internal messages should be able to perform any necessary message transformation flow.

These characteristics represent the basic functionality of an ESB. They are mandatory to provide the functionality needed to establish enterprise integration with the architectural benefits of an Enterprise Service Bus [Cha09].

This thesis will describe the routing logic inside an ESB as message flows or mediator flows. According to Bobby Woolf[2] ESB routing implements mediation flows but no business processes. Workflows can use an ESB but typically run in a workflow engine.

2.3 Enterprise Topology Graphs

The structure of the enterprise IT is very complex and often causes a loss of insight into the enterprise topology. This complexity of the IT landscape covers not only the hardware topology but also extends to all entities of an enterprise IT, including their logical, functional, and physical relationships.

[2]DeveloperWorks Blog of Booby Woolf - https://www.ibm.com/developerworks/community/blogs/woolf/entry/esb_and_workflow?lang=en

All these elements spread over different levels of abstractions which can be modeled with an Enterprise Topology Graph [BFL$^+$12]. The standard is mainly influenced by the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [Top11][BBLS12]. The flexible graph-based structure of the ETG makes it possible to model an application with arbitrary composition and structure. Thus the advantage of this approach is that a formal graph enables processing of the complete enterprise architecture. This should solve the lack of available machine-readable enterprise topology and could establish new requirements and market demands, integration and IT consolidation.
The Enterprise Topology Graph model consist of a set of nodes and edges. These entities are typed with domain specific semantics that can be used to establish different kind of abstraction. Every part of an enterprise topology can be assigned as a node, e.g an application server or operating system. However these elements of the enterprise IT are linked together as trees. These links are basically relations between nodes modeled by directed edges with a specific type.
The ETG should contain all information that is available to satisfy the demand of a complete snapshot of the enterprise IT landscape. Consequently of this all steps of the routing in an Enterprise Service Bus should be documented and represented in the graph [BFL$^+$12].
The left side of Figure 6.2 representing a running Apache Synapse ESB displayed in the ETG model. It depends on ApacheAxis2 Web server hosted on the ApacheOS that is running on an virtual machine. This could only describe a segment of the complete enterprise IT. Both nodes Server and Operating System are connected by an edge with the node type *hosted on*. The same relation exist between the operating system and the ApacheAxis2 Web server. Apache Synapse depends on the Web server because it will receive the request messages from it. The nodes can also capture domain-specific information as key-value pairs as properties. These information can be processed with the context of the specific node type [BLNS12].
This sub graph only consists of three nodes and two edges. But in a real scenario a graph may consist of millions of nodes. However, this complexity of the enterprise architecture has to be reflected in the ETG. But the model has to cope with requirements like queries, transformation and consolidation in the data format. Associated with the ETG model, there are approaches which reduce these complexities problems. That address the current problems of analyzing an IT infrastructure. The ETG Model has been implemented with the current ETG Framework [BFL$^+$12].

2.4 ETG Framework

Enterprises often have no comprehensive view of the structure of their IT landscape [BFL$^+$12]. The main problem is the complexity and the diversity in the topology of countless components. The manually maintenance of a huge topology could only be realized with a high level abstraction. With this simplification a Content Management System can be abstractly represented as one node in a the ETG graph. But the dependencies and relationships of the components would be lost due to this abstraction. The other option is an automated discovery that reveals the inner structure of complex architecture components [BLNS12].
The ETG Framework implements an automated discovery of enterprise IT [BBKL13]. The

implementation works like a search engine crawler, with component specific logic provided as plugins. The frameworks architecture invokes specialized plugins with a scheduling algorithm. The scheduler decides when and which plugin will be executed in what node. The execution is an iterative process with a step wise discovery of the IT landscape of an enterprise. Each step in the iteration will execute a compatible plugin that matches the given type of the previous discovered node. For instance, a server node will be examined which operating system is running on it. There are also modifying plugins which change the type or properties of an element. In a nutshell the basic strategy of the scheduler is to pick a compatible plugin only once for every node and terminate if every node was processed.
The previous mentioned plugins are used to provide the discovery logic. They are designed for a specific type in the ETG model that represents a functionality or component in the IT structure. The plugin gathers all information like environment, dependencies and relation details of a node. The plugins also update the nodes and edges or even remove them if they are stale, like a node has been shut down or has been reconfigured.
The plugins are even capable to work together. For instance an plugin would discover a web service that is hosted on an undiscovered application server. The web service plugin would create a new node that would be picked up by application server plugins in a future iteration step by the scheduler. Plugins cannot provide agents that run on the enterprise servers and possible scan for installed software in the domain.

The ETG Framework is implemented by using the approach for discovery and a reconciliation concept using Java technology. Furthermore, there are already several working plugins within the framework.

2.5 Apache Camel

Integration is the key to accomplish the usability of existing Software artifacts of today's IT landscape. Hohpe and Woolf summarized theses approaches in their work. They stated 65 different enterprise integration patterns as help for architects and developers to design a proper integration solution [HW02]. These patterns represent the theoretically framing for a eligible software solution to empower engineers to integrate their products. The first version of Apache Camel was released in 2007 and provided several endpoint implementations and an easy to use Domain Specific Language for creating Enterprise Integration Patterns[3] and routing. Today Apache Camel is a sophisticated integration framework with a large open-source community. The Release 2.7 provides a rich set of features. The important characteristics for this diploma thesis with the focus on the ESB aspects are as follows:

- Routing and mediation engine
- Payload-agnostic router
- Enterprise Integration Patterns

[3]Apache Camel Release Notes - https://issues.apache.org/jira/secure/ReleaseNote.jspa

Claus Ibsen and Jonathan Anstey accentuate that Apache Camel is not an Enterprise Service Bus because of the absence of a container or a reliable message bus. But this functionality can be extended with products like Apache Active MQ or the OSGi framework [The13]. These features are bundled in a distribution called Apache ServiceMix. However the core functionality is provided by an unchanged Apache Camel implementation. Guillaume Nodet[4] argued in a discussion about Apache ServiceMix that users tend to start their project directly with Camel without the feature overhead of the ServiceMix. This thesis will follow the same approach and focus only on the Apache Camel framework and will enhance the framework with features from the Apache ServiceMix if needed. In a nutshell Apache Camel is a lightwight Enterprise Service Bus and can be expanded with features from other open source products. It is easy to use, easy to deploy and comes with a large and vibrant community. The core features are routing, transforming, usage of Enterprise Integration Patterns, management and monitoring [IA10].

2.6 Apache Synapse

The roots of Apache Synapse go back to the year 2005. The project started by several cooperating small companies in the Bay Area. The first release in the year 2006 is still the codebase of the current Apache Synapse release. Finally, in the late 2007 the project was leveled as an Apache top level project. Since then there were 3 major releases [Thed]. Today, Apache Synapse can be downloaded and used under the Apache Software License 2.0. The Documentation [Thee] describes Apache Synapse as an extremely lightweight and easy-to-use Open Source Enterprise Services Bus. This definition implies Enterprise Service Bus characteristics like:

- Routing and mediation Engine
- Runtime container
- Messaging
- Support of different transport Formats like HTTP, Mail and JMX.

The Framework can be configured by using a simple, XML-based configuration language. According to the documentation, the user can choose between different functional components designed with SOAs best practice in mind [Thed]. The portfolio of mediators contains filter, transformation and routing Elements, that can indeed be compared with Enterprise Service Patterns. Synapse can be run as a standalone Java application or in a Tomcat Application Server.

[4]Guillaume Nodet post- http://servicemix.396122.n5.nabble.com/DISCUSS-ServiceMix-future-td3212177.html

2.7 Information Gathering Strategies

Software analysis is the primary research subject of software maintenance. In maintenance it is most important to have an accurate, up-to-date and useful knowledge about the system that is being maintained. This is a difficult task because the sources of information are often limited, inaccessible or unknown [Sea02].
This thesis faces the same difficulties and we need to consider good strategies that can be used to gather information from an ESB. Therefore the different routing information can be categorized as follows:

- **Static routing information**
  The description of the internal execution progress of the defined routes. They can have a proxy endpoint that exposes an address to the outside. Followed by optional mediators and gateways with route condition elements, finally targeting an endpoint that serves the request.
- **Statistical routing information**
  These are statistical information that only accrue while the ESB is running. The received messages of an endpoint is such a value.

There are only several gathering techniques viable for the analyses of the routing of Enterprise Service Bus middleware. There are several possible static sources of routing data that ESB can offer:

- **Log files**
  Java application usually generate log files during execution. They could be analyzed for routing information.
- **Routing configuration files**
  Apache Synapse uses external configuration files as XML to setup the ESB. These files are accurate sources for static routing information.
- **Source code of the application**
  The source code of an mediation flow scenario could be parsed and analyzed.
- **Machine readable ESB scenario descriptions**
  Several ESB like Apache Camel are based on Enterprise Integration Patterns. They are useful to develop integration scenarios into an ESB middleware system. The Genius Tool has implemented a parameterized EAI pattern called the EMF model. This can be transformed to BPEL and Apache Camel. An platform independent model could be used as information source for an ESB analysis [SL09].

Not all of these options can be considered as source of the plugin information gathering. The inspection of the source code does not match with the requirement of the ETG Framework, that a plugin should require minimized operational impact [BBKL13]. And the plugin cannot determine if there is a machine readable ESB scenario description accessable. So only log files and routing configuration files are practical for retrieving static information from an ESB.
Hence, the statistical information cannot be obtained in all of the previous mentioned sources.

Dynamic information is generated on an running ESB system. So the information has to be produced or accessed through the application. There are three possible sources to get access to the information:

- **Log files**
  Log files could register the message flow in the application. This could be used to reassemble the needed statistics.
- **Custom ESB management API**
  The Enterprise Service Bus could support a management API that can be used to access information.
- **Java Management Extensions (JMX)**
  The Java Management Extension Technology was introduced in the J2SE platform 5.0 and is now included in the Java SE platform. It is designed for management and monitoring of resources. JMX can change application configuration and accumulation statistics about behavior that can be made available. It is also possible to access these information remotely [ORA11].

The ETG Framework plugin developer must choose the most applicable solution for the information gathering. Therefore, the choice should be reviewed within a new release if better options are available to ensure minimized operational impact and quality [BBKL13].

Chapter 3

Focus on Apache Camel

This Chapter focus on Apache Camel. In Section 3.1 the ESB will be classified using characteristics from David Chappell. Followed by the architecture and concepts of Apache Camel in Section 3.2. Yet, the focus of this Chapter is to gather static routing information and statistics from an ESB. Section 3.3 analyses the structure of the routing information. Apache Camel is widely used as ESB implementation and Section 3.4 presents two use cases. For the purposes of this thesis, several technologies are needed to enhance Apache Camel they are documented in Section 3.5. The Apache Camel scenarios in Section 3.6 are used to evaluate the information gathering methods introduced in Section 3.7. Finally, all the research has been implemented as Apache Camel information crawler. The technical details are discussed in Section 3.8 and the results will be discussed in Section 3.9.

3.1 Classification

Apache Camel is designed as an easy to use open source integration framework. The core of the framework is a routing engine that orchestrates the dispatching and receiving of messages. The transport protocol is not part of the core of Apache Camel and users can freely choose a protocol that suites their architectural environment best. It is even possible to use different protocols for message channels between endpoints. These can freely be utilized without dealing with specific implementation efforts.
The higher-level abstraction of Apache Camel decouples the route configuration with the transportation between a receiver and sender. The participants in the message interaction are also abstractly defined as Apache Camel endpoints entities that encapsulated the message exchanges as abstract processors. The message flow is facilitated with channels between two or multiple endpoints. Thereby Apache Camel is using a unified communication API regardless of the transport and request format of the message.
These concepts build the core functionality of the Camel Framework that implements the ESB characteristics. The messaging capability is provided by Apache MQ messaging broker. These extensions are necessary to provide a capable ESB middleware [IA10] [The13]. Table 3.1 compares the ESB characteristics defined by David Chappell in 2004 with the Apache Camel

ESB Characteristics	Apache Camel functionality
Pervasiveness	Every Java Application can simply be attached as a Camel endpoint. Adapters for the conversion of data types are present. Every endpoint can be reached within the same *camel context*.
Standards-Based Integration	Camel provides an out-of-the-box support with over 60 transport protocols. Applications based on different technologies can be attached with a custom made Java adapter. Internet based standards like SOAP and WSDL are supported.
Highly distributed, event-driven SOA	The message exchange is orchestrated over a central Apache Camel instance. This can be extended on multiple servers with different *camel context*. But this limitation is connected with the flow engine and orchestration feature of Apache Camel. Asynchronous communication can be used as transportation protocol (e.g. Apache MQ).
Security and reliability	The camel-spring-security ensures security for routes. Also there are broad categories offered like route, payload, endpoint and configuration security. There are components that enhance Apache Camel route with reliability like the SEDA component (cf. section 3.5.5).
Orchestration and process flow	Orchestration is implemented as a routing engine in the core of the application. A process flow can be built with the Apache Camel pattern system.

Table 3.1: Comparison of ESB characteristics and Apache Camel functionality

functionality. Apache Camel meets the requirements discussed in Chapter 2.2 formalized as characteristics of an Enterprise Service Bus [Cha09].

3.2 Architecture and Concept

The concepts and architecture of Apache Camel are fundamental for solving the unknown routing problem. The Apache Camel routing rulebase is programmatically represented as the *camel context*. It centralizes the access to all orchestrated components, endpoints and routes. The camel context instance also represents the life cycle of the application. It controls the different Camel entities and routing rules from the start of the context to its terminations [The13].
ESB flows can be composed using messaging patterns and Web services. A single operational task or logical functionality can be built as a route. Thereby a mediator flow is often crafted

by combining several predefined Apache Camel routes.
Usually an endpoint can be created by a component that is referenced with their URIs. The instance of an endpoint, which represents the giving work of the activity, can be very versatile. Depending on the used Apache Camel component the exposed transportation type can vary. The transportation logic used by Apache Camel is auto-discovered using the prefix of the endpoint address URI. Endpoints can be accessed using several transport protocols and access methods. The type of component of an endpoint can vary:

- Web service component using a WSDL port.
- A messaging queue like Apache MQ, Websphere MQ or a JMS queue with a polling backend application.
- Direct access to a software entity
- File component
- FTP server component
- E-Mail component

The message exchange is defined by the Producer and Consumer interfaces associated with the endpoint implementation. The messages can be received either as polling or pulling consumer.
This illustrates the wide possibilities to connect several services to the Enterprise Service Bus. The abstraction does not make any assumptions about the used transport protocol. Surprisingly, even an e-mail address can be used as an Endpoint. The given service would be invoked with an incoming e-mail from a unknown decoupled client service. This transport and processing step is out of the Apache Camel scope, Camel just needs access to the endpoint features given in the corresponding processor interface.
Besides, the processor interface implementation describes the exchange of messages within a Apache Camel node and the outside. Every enterprise pattern refers to an implementation of a processor exchange functionality. Also the message exchange patterns are using a custom implementation of the processor interfaces. For instance the routing pattern that usually is prepended in front of multiple endpoints and acts as a content-based, load <balance or static router. All in all, the processor controls the consumption and exchange of messages of every Apache Camel entity.
The previous mentioned component paradigm is a central concept of Apache Camel architecture. The component acts like an Java factory method pattern that creates objects of a specific type of endpoint. The used component implementation enables Apache Camel to integrate applications using different protocols.

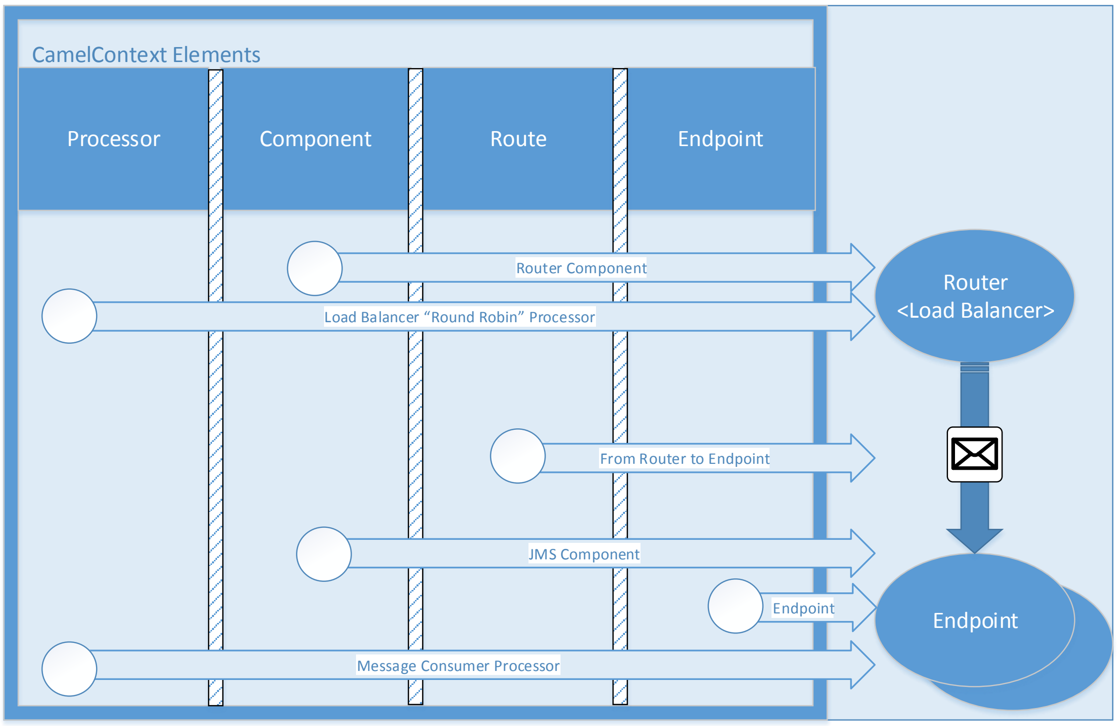


Figure 3.1: Illustration of the Main Components of Apache Camel building an example route

With this technique the endpoint can be automatically invoked by interpreting a URI, whose prefix indicates the type of transportation and the address or other information parameters. There are already mandatory Apache Camel components available, for example the *FileComponent* that gets an URI passed with a prefix file and an appended URL to the destination. Then Apache Camel creates an *FileEndpoint*.
These Apache Camel nodes are organized into a defined route pattern. This message chain defines the invoked mediators on the message flow. Each node is connected by channels that are used for transportation of a message. A node can also provide decision making, if it acts as a filter or router. There are two ways to specify a route either direct in the code through the camel context object or via a Java Domain Specific Language in a XML file.
Figure 3.1 shows the key components in correlation with a possible camel context. On the right side of the Figure there is a sample route from a router to a JMS endpoint. The router is called a loadbalancer router (simplified in the picture) and distributes messages with a *round-robin* algorithm. The abstraction makes it easy to define such an orchestration between different entities and endpoints. This is the key concept of camel that you can combine every piece of component and build a route to establish the defined business logic.
Apache Camel is a universal tool for enterprise integration challenges. The concept of combining different enterprise patterns and endpoints with different processors and transport protocols using the component factory enable flexible application integration.

3.3 Pattern Overview and Structure

The subsections discussed concepts and components that can be allocated to defined patterns in Apache Camel. Now this section covers the patterns that are necessary for the interpretation of the camel structure crawler. The complete list of the available patterns is documented in the Apache Camel manual [The13].
The Camel JMX output has been collected using the output of the camel crawler. This section categorizes Apache Camel patterns with the crawled JMX information from the deployment scenarios discussed in Section 3.6. Unfortunately the JMX output is unstructured and represented as string. There is no documentation of the precise syntax of the JMX description of the routes. Each subsection will show a table with the enterprise integration pattern, a short description and the JMX output [The13].
This categorization helped to re-engineer the description syntax. The crawler needs this syntax to analyze the components of a Apache Camel route. The normalized structure of the output can be compared in Listing 3.1.

Listing 3.1 The syntax of the Apache Camel JMX output

```
routeType[endpoint->Instrumentation:route[transmissionSynchronization]
[UnitOfWork(ProcessorType[Channel/Pipeline[PatternType[endpoints*]])])
```

The structure of the JMX string can be analyzed with a string parser iterating the String structure from left to right. The semantics and rules for each item are defined as follows:

1 **routeType**
Typically an EventDrivenConsumerRoute. That represents the default route type in Apache Camel.

2 **endpoint**
That represents a proxy endpoint which receives a message from a client.

3 **->Instrumentation:route**
JMX metadata can be ignored.

4 **transmissionSynchronization**
Usually DelegateAsync or DelegateSync messages processor for the routing pattern invocation.

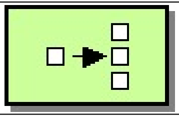
5 **UnitOfWork**
JMX metadata signal word introducing the start of mediator patterns and endpoints.

6 **Channel/Pipeline**
Either a Channel or an Pipeline to the next patternType

7 **PatternType[endpoints*]**
The *patternType* followed by zero or multiple endpoints.

The Camel crawler implementation can use this knowledge and parse the objects contained in the string. The following pattern catalog is used to illustrate the JMX output. They are the backbone information for the re-engineering process. The result is the decoded syntax.

3.3.1 Splitter

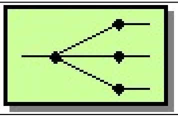
	Splitter [HW02]
Description	A Message can be composed of multiple logical different parts. The handling of these parts could need a isolation of the information. The splitter produces a new message for each element.

Camel route DSL	`<route>`
	    `<from uri="direct:cafe"/>`
	    `<split>`
	        `<method bean="orderSplitter"/>`
	        `<to uri="direct:drink"/>`
	    `</split>`
	`</route>`

Camel JMX Output	EventDrivenConsumerRoute[Endpoint[direct://cafe] -> Instrumentation:route[DelegateAsync[UnitOfWork(Route ContextProcessor[Channel[Splitter[on: BeanExpression[] to: Channel[sendTo(Endpoint[direct://drink])] aggregate: null]]]])]]]

Table 3.2: Apache Camel splitter

3.3.2 Recipient List

	Recipient List [HW02]
Description	An incoming message will be sent to a selected list of end-points.

Camel route DSL	`<route>`
	    `<from uri="direct:drink"/>`
	    `<recipientList>`
	        `<method bean="drinkRouter"/>`
	    `</recipientList>`
	`</route>`

Camel JMX Output	EventDrivenConsumerRoute [EventDrivenConsumerRoute[Endpoint[direct://drink] -> Instrumentation:route[DelegateAsync[UnitOfWork(RouteContextProcessor[Channel[RecipientList[BeanExpression[]]]])]]]

Table 3.3: Apache Camel recipient list

3.3.3 Pipes and Filters

	Pipes and Filters [HW02]
Description	The processing of a task could require a sequence of smaller independent tasks.

Camel route DSL	`<route>`
	`    <from uri="seda:coldDrinks"/>`
	`    <to uri="bean:barista?method=prepareColdDrink"/>`
	`    <to uri="direct:deliveries"/>`
	`</route>`

Camel JMX Output	EventDrivenConsumerRoute[Endpoint [seda://coldDrinks?concurrentConsumers=2] -> Instrumentation:route[DelegateAsync [UnitOfWork(RouteContextProcessor[Pipeline[[Channel [sendTo(Endpoint [bean://barista?method=prepareColdDrink])], Channel[sendTo(Endpoint[direct://deliveries])]]]])]]]

Table 3.4: Apache Camel pipes and filters

3.3.4 Aggregator

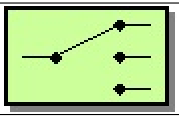
	Aggregator [HW02]
Description	The Aggregator stores correlated messages until all have been collected. Then a single message will be consolidated as a result of the complete set of small messages.

Camel route DSL	<from uri="direct:deliveries"/>
	<aggregate strategyRef="aggregatorStrategy" completionTimeout="5000">
	    <correlationExpression>
	        <method bean="waiter" method="checkOrder"/>
	    </correlationExpression>
	    <to uri="bean:waiter?method=prepareDelivery"/>
	    <to uri="bean:waiter?method=deliverCafes"/>
	</aggregate>
	</route>

Camel JMX Output	EventDrivenConsumerRoute[Endpoint[direct://deliveries] -> Instrumentation:route[DelegateAsync[UnitOfWork (RouteContextProcessor[Channel[AggregateProcessor [to: UnitOfWork(RouteContextProcessor[Pipeline[[Channel [sendTo(Endpoint [bean://waiter?method=prepareDelivery])], Channel[sendTo(Endpoint[bean://waiter?method= deliverCafes])]]]])]]])]]]

Table 3.5: Apache Camel aggregator

3.3.5 Message Router

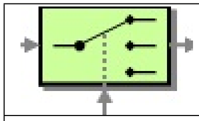
	Message Router
Description	The static message router consumes a message and is redirecting the request to an endpoint [The13].

Camel route DSL	`<route>`
	    `<from uri="jetty:http://localhost:port/Cx/Port"/>`
	    `<loadBalance ref="roundRobinRef">`
	        `<to uri="direct:CalcEndpointA" />`
	        `<to uri="direct:CalcEndpointB" />`
	        `<to uri="direct:CalcEndpointC" />`
	        `<to uri="direct:CalcEndpointD" />`
	    `</loadBalance>`
	`</route>`

Camel JMX Output	EventDrivenConsumerRoute[Endpoint [http://localhost:9001/Cx/Port] -> Instrumentation:route[DelegateAsync[UnitOfWork(RouteContextProcessor[Channel [RoundRobinLoadBalancer]])]]]
Possible Routing-Policies	RoundRobin, Random, Sticky, Topic, Failover, Weighted Round Robin, Weighted Random and Custom

Table 3.6: Apache Camel message router

3.3.6 Dynamic Router

	Dynamic Router
Description	It provides message routing based on a dynamic Routing-Policy to a suitable endpoint. The Routing policy is defined in a custom Java bean. [The13]

Camel route DSL	`<route>`
	`    <from uri="direct:start"/>`
	`<dynamicRouter>`
	`        <method ref="myBean" method="route"/>`
	`    </dynamicRouter>`
	`    <to uri="mock:result"/>`
	`</route>`

Camel JMX Output	EventDrivenConsumerRoute[Endpoint[direct://start] -> Instrumentation :route[DelegateAsync[UnitOfWork(RouteContextProcessor [Pipeline [[Channel[RoutingSlip[expression=BeanExpression [method: route] uriDelimiter=,]], Channel[sendTo(Endpoint[mock://result])]]]])]]]
Parameter	uriDelimiter and ignoreInvalidEndpoints

Table 3.7: Apache Camel dynamic router

3.4 User Stories

3.4.1 Apache ServiceMix

The product description [Thei] promotes ServiceMix as a flexible, open-source integration container that unifies the features and functionality of Apache ActiveMQ, Camel, CXF, ODE, Karaf into a powerful runtime platform. The framework can help developers to create their own integration solutions. It is basically a ready to go distribution of an Enterprise Service Bus. The package includes complete messaging, routing and enterprise integration functionality powered by an unmodified Apache Camel. The integration of other packages from the Apache Foundation enhance the functionality of the ESB:

- **Reliable messaging with Apache MQ**
  Camel provides a direct component in their core libraries [Theg]. It allows synchronous invocation of messages by sending from a producer to an consumer directly. It is a simple transport protocol between Java service beans. The Apache MQ project supports many cross languages clients and protocols, as well as full support for the Enterprise Integration Patterns and a sophisticated Message Broker [Thea].
- **WS-* and RESTful web services with Apache CFX**
  Apache CFX is a framework that provides a frontend API that enables an easy creation of Web services based on standards like SOAP, XML/HTTP, RESTful HTTP, or CORBA [Theb].
- **Complete WS-BPEL engine with Apache ODE**
  With Apache ODE it is possible to write business processes with the XML-based language defined as WS-BPEL. The framework relies on WSDL to express Web service interfaces and supports long and short living process execution [Thec].
- **OSGi-based server runtime with Apache Karaf**
  The Apache ServiceMix is enhanced by Apache Karaf a lightweight container for various applications. These can be hot deployed, controlled and managed.

The distribution of Apache Service Mix is a prepackaged ESB all-in-one solution for enterprises. But it is also possible to selectively enhance Apache Camel with the desired features.

3.4.2 Fuse ESB Enterprise

Fuse ESB is a supported distribution of Apache ServiceMix. They are integrated, tested and supported with the reliability and expertise of commercially provided enterprise services. The ESB product also provides IT organizations with installers, support for patching and verification for security flaws. The developing company Red Hat contributes with over 25 active developers under the Apache License. They introduce extension for the service mix for example, JBI container for JBI artifacts or user tools helping to create routes from a diagram to implementation [Red].
Fuse ESB become accepted in the industry. At this point one should not forget to mention two user stories:
Hermes is a logistic enterprise that is the largest home deliver courier in the United Kingdom. The company wanted to avoid a vendor lock-in but was also not comfortable with a pure open source due to the lack of support. With the Fuse ESB they could use an supported and consulted Apache service mix distribution and they could get a grip on some of its problems like real time delivering information and increased control and monitoring of the business [Red12].

Another example of a successful user story is a warehouse management system built with Fuse ESB for a major retail pharmacy chain. This warehouse management solution should eliminate manual scheduling while limiting the costs of operation. The delivery of products is scheduled to avoid idling of trucks. All existing applications had to be interfaced including third-party systems and legacy mainframes. Within a short time they could remodel the whole system using Fuse ESB. This was done without major implementation effort [Theh].

3.5 Camel Technologies

As mentioned before, Apache Camel does not come prepackaged with the components needed in this thesis. This section will briefly introduce some key technologies used with Apache Camel. They are used for the implementation of the Apache Camel examples (cf. Section 3.6) or in the Apache Camel crawler (cf. Section 3.8). They are basically third-party program providing key functionality. The following components are arranged in alphabetical order and the content focuses on the usage.

3.5.1 Apache CFX

Apache Camel enables Web service support as an Apache Camel CFX component. It provides the integration of protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA and enables the connection to JAX-WS services. Using Apache Maven the needed dependencies can be imported adding the following artifacts: the transport (*cxf-rt-transports-http-jetty* or *cxf-rt-transports-jms*) and SOAP bindings (*cxf-rt-bindings-soap*). The CFX component can easily be instantiated as Apache Spring bean. Then the *cxfEndpoint* represents the bean ID in the Spring bean registry [The13].

3.5.2 Direct Component

The direct component is a transmission channel for messages. It provides an easy direct message exchange inside of Apache Camel. But it is not suitable for asynchronous communication. It has often been used to link two routes together. You can also invoke a route from an accessible Java class that can access the *camel context* object. It is good practice that the start endpoint (more precisely the proxy cf. section 5.2.2) is called "direct:start".

3.5.3 Jetty Component

Jetty is the only Apache Camel supported web server that is standards-based and available under the Apache License 2.0[1] and Eclipse Public License 1.0[2]. The project also provides a HTTP client and an *javax.servlet* container. The container features JNDI, JMX and session management. The client and server supports highly scalable asynchronous communication [Jet12]. The performance of Jetty is not well suited for static content. However it performs

[1]Apache License 2.0 - http://www.apache.org/licenses/LICENSE-2.0.html
[2]Eclipse Public License 1.0 - http://www.eclipse.org/legal/epl-v10.html

better with dynamic content generating. It is robust under overload and is capable for emerging Web technologies like cloud computing [TAW03].

3.5.4 SEDA Component

Camel provides asynchronous staged event-driven architecture (SEDA) behavior. All messages are stored in a *BlockingQueue* that provides thread safety between consumers and producer. The SEDA concept is designed for highly concurrent Internet services. The architecture is event-driven and can be used for massive concurrency demands. That is helpful because an ESB often faces huge loads of requests from clients. Request with dynamic context creates more messages and automatically lead to massive I/O and computation.
There is a lack of concurrency support in traditional operating system designs. Also the widely promoted models on concurrency do not provide a sufficient management of load. SEDA solves this issues and serves as general framework for concurred service instance [WCB01].
The SEDA queue can be facilitated within a single *camel context*. The usage of the queue is very simple. The user must not define the SEDA bean inside the Spring configuration. On default the component will be instantiate as *LinkedBlockingQueue*. The SEDA queue can be addressed in the *camel context* just by using URI format *seda:someName*.

3.5.5 Spring Framework

Apache Camel heavily utilizes the Spring Framework in their application concept and is also capable to support several Spring features like:

- **Spring Transactions**
  Transactions can be applied to the message flow of Apache Camel.
- **Spring XML Processing**
  The *camel context* can be invoked in a spring XML configuration.
- **Spring beans**
  Beans can be designed in a Spring application context. Spring beans act like a container that produces a fully configured system which is ready for use. The user configures the Spring beans with XML-based metadata in the corresponding Spring XML file. The beans can be instantiated in the camel route XML or directly in the Java code [Spr13].
- **Integration of various Spring Helper classes**
  Camel provides a type converter that support Spring resources.
- **Spring dependency inject component**
  This feature enables an external definition of the *camel context* as XML file. The corresponding objects like the routes themselves, will be automatically exposed as components and endpoints.

3.6 Evaluation Scenarios

3.6.1 Load Balancer

Szenario:

This example deploys one proxy endpoint that is consuming a SOAP message over a client HTTP request. There are four redundant endpoints that host the same implementation of a service.
The scenario of this example is a calculator application acting as Apache Camel client and performing add, subtract, multiply and divide operations against the proxy. Each endpoint implements these basic mathematical functions and return the result. Those Web services will be hosted simultaneously in an Apache Jetty server instance. The calculator application addresses the calculation requests to the ESB while acting like a normal Windows calculator for the user. There is also an exception handling implemented for division by zero violations. The exceptions will be raised on the backend services, hosted on Apache Camel and finally handled in the user-application. This whole functionality represents a complete Web service powered client application with an Apache Camel configuration that could handle a huge amount of requests.

Architecture:

The calculation service implements the calculator interface. This interface is being derived from the corresponding WSDL file.

Listing 3.2 Service interface of the Calculator WSDL file

```
@WebService
public interface ICalculator {
      public Result[] addition(@WebParam(name="numberA") double numberA,
          @WebParam(name="numberB") double numberB);
      public Result[] substraction(@WebParam(name="numberA") double numberA,
          @WebParam(name="numberB") double numberB);
      public Result[] multiplication(@WebParam(name="numberA") double numberA,
          @WebParam(name="numberB") double numberB);
      public Result[] division(@WebParam(name="number") double numberA,
          @WebParam(name="devisor") double numberB) throws NaNException;
}
```

The code in Listing 3.2 illustrates the operations provided by the service. The return class encapsulates the result and will be serialized to the client. The result objects contain the result value of the operation and also have a correlation ID enabling the client to map the calculation outcome with an application context like an equation. As discussed a division by zero causes a *NaNException* that will be thrown. These classes are the input for the *cxf-java2ws-plugin* which builds the WSDL file and the Web service Interface.

The calculation server creates a *SpringBusFactory* that instantiate the Apache Camel route XML file which automatically setups the *camel context*. The server publishes the Java implementation of the backend web service endpoints. This represents the integration of calculation applications exposed as an Web service.
The calculator client implementation facilitate the communication using the WSDL port. Using the proper XML binding it enables the communication with the endpoint. The message serialization has been automated using CFX annotation. Finally the calculator application can invoke the function *division(double number, double divisor)* like a local Java function. All the complexity is hidden using Apache Camel with Apache CFX.

Listing 3.3 Example of one endpoint configured as jaxws endpoint

```
      <jaxws:endpoint id="calcEndpointA" implementor="#calculationA"
            address="camel://direct:CalcEndpointA" />
```

Consequently the configuration of the endpoints using the *jaxws* endpoint (cf. Listing 3.3) declaration that provides the Web service functionality and using the *JAX-WS-APIS* interface that enables the CFX support.

Camel Context:

Listing 3.4 Calculator example route using a load balancer with four endpoints

```
<route>
      <from uri="jetty:http://localhost:{{port}}/CalculationContext/CalcPort" />
      <loadBalance ref="roundRobinRef">
            <to uri="direct:CalcEndpointA" />
            <to uri="direct:CalcEndpointB" />
            <to uri="direct:CalcEndpointC" />
            <to uri="direct:CalcEndpointD" />
      </loadBalance>
</route>
```

Listing 3.4 illustrates the *camel context* of this example. The Client can invoke the route using the *jetty:http://localhost:port/CalculationContext/CalcPort* URI. The client request message will be distributed to an calculation endpoint using a *round robin* algorithm. The *round robin* algorithm can be considered as fair and can execute in constant time. In this example we can ignore the packet sizes because the calculator class file will be serialized by Apache CFX. The difference would be so small that there is no need to implement a deficit *round robin*. The different four calculation task of the services have no high computation complexity. So there is no need for queuing based on the load [SV96]. The classical *round robin* is very suitable for this example.

Expected Crawling Result:

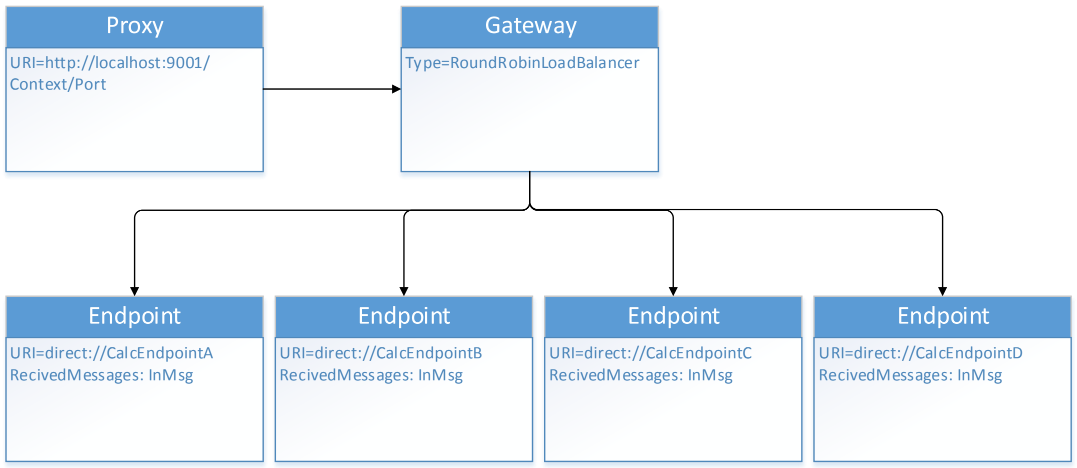

Proxy
URI=http://localhost:9001/Context/Port
Gateway
Type=RoundRobinLoadBalancer
Endpoint
URI=direct://CalcEndpointA
RecivedMessages: InMsg
Endpoint
URI=direct://CalcEndpointB
RecivedMessages: InMsg
Endpoint
URI=direct://CalcEndpointC
RecivedMessages: InMsg
Endpoint
URI=direct://CalcEndpointD
RecivedMessages: InMsg

Figure 3.2: Structure of the load balance example evaluation

Figure 3.2 illustrates the evaluated crawling result. The crawler has identified all corresponding endpoint nodes. The result is consistent with the generic ESB data model of Ssection 5. The crawler plugin returned the correct structure stored in the *CamelStructure* object. The strings analysis method discussed in Section 3.3 has been used.

Summarized the crawler has detected six generic ESB model entities:

- One Proxy node that addresses the route.
- One Gateway
  type = RoundRobinLoadBalancer
- Four Endpoints
  each connected to the gateway, all providing different internal endpoint URIs.

3.6.2 Business Example - The Coffee Shop

Szenario:

The Apache Camel cafe example[3] will be used to illustrate the collaboration between different Apache Camel components and represent a standard scenario for an Apache Camel mediator

[3]http://camel.apache.org/cafe-example.html

flow.
The example describes a coffee shop use case that receives orders and then prepares the desired drinks for the client. First the client sends an order to a proxy representing the start of the Apache Camel route. An order usually includes several items. Every item will be processed individually. The shop offers cold and hot drinks, which are prepared from different entities. The waiter delivers the coffee after all items of the order are ready.

Camel Context:

Listing 3.5 First coffee shop route from *direct:cafe* invoking a splitter then send to *direct:drink*

```
<route>
  <from uri="direct:cafe"/>
  <split>
    <method bean="orderSplitter"/>
    <to uri="direct:drink"/>
  </split>
</route>
```

First Apache Camel route illustrated in Listing 3.5 declares the proxy endpoint that receives messages from clients. It forwards the messages to a splitter mediator. The splitter implementation is provided by an *orderSplitter* bean.
The Order contains a list of one or more drinks each now referred to as items. Each item will be isolated and sent to the next route.

Listing 3.6 Second coffee shop route from *direct:drink* invoking a *recipient list* bean

```
<route>
  <from uri="direct:drink"/>
  <recipientList>
    <method bean="drinkRouter"/>
  </recipientList>
</route>
```

In the next step in Listing 3.6 each item will be redirected to a recipient list that acts like a content router. The *drinkRouter* bean routes the message depending on the context of the item to a suitable endpoint.
The recipient is choosen based whether the item equates as cold or hot drink. The cold drinks will be routed to "seda:coldDrinks" and the hot drinks to "seda:hotDrinks". The routing alternatives are hidden from the crawler. The implementation of the *drinkRouter* bean routes the message outside of the *camel context*. Both drink routes, targeted by the *drinkRouter*, pulls messages from a SEDA (cf. section 3.5.5) queue. The address of the queue is defined in the from element with the URI attribute.

Listing 3.7 Coffee shop preparation route alternative A for cold drinks

```
<route>
  <from uri="seda:coldDrinks?concurrentConsumers=2"/>
  <to uri="bean:barista?method=prepareColdDrink"/>
  <to uri="direct:deliveries"/>
</route>
```

The bean in Listing 3.7 has a load limit of two concurrent customers. This illustrates a typical option that can be set in the SEDA component. After the preparation of the cold drink, the *direct:deliveries* route will be invoked.
The second alternative is related to the previous route with the exception of preparing hot drinks and accepts three concurrent customers.

Listing 3.8 Final route aggregating the orders, preparing and finally deliver the drinks

```
<route>
  <from uri="direct:deliveries"/>
  <aggregate strategyRef="aggregatorStrategy" completionTimeout="5000">
    <correlationExpression>
      <method bean="waiter" method="checkOrder"/>
    </correlationExpression>
    <to uri="bean:waiter?method=prepareDelivery"/>
    <to uri="bean:waiter?method=deliverCafes"/>
  </aggregate>
</route>
```

Finally the order-items are directed to the *direct:deliveries* route illustrated in Listing 3.8. The aggregation pattern collects all items until all items of one order have been delivered. The waiter bean can investigate the order number of an item. This will be used as correlation expression for the *aggregationStrategy*. If the complete condition was meet, the order will be delivered using the *bean:waiter?method=deliverCafes* bean. This bean prints the final prepared order on the console.

Expected Crawling Result:

The crawler does not assume any correlation between the routes. This limitation of the crawler will be discussed in section 3.9. This example creates an indirection between the routes. The hidden router implementation in the *drinkRouter* causes unavoidable disconnections in the logical chain of routes. The evaluated crawling results are following in the same scenario appearance using the notion of the generic ESB data model:

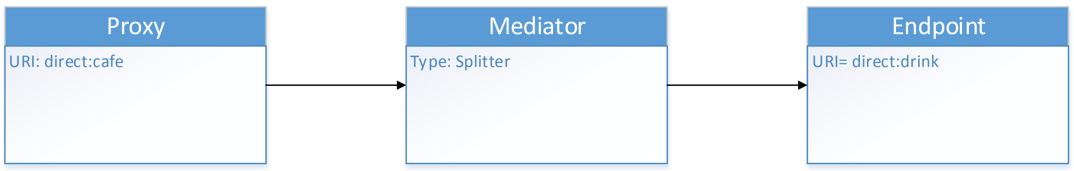


Figure 3.3: Structure of the expected splitter route elements

The first route in Figure 3.3 represents the crawling result that splits the order in order-items. The splitter is not a gateway because it can only have one successor and acts like a mediator.

Generally a list of recipients can define a multicast among defined endpoints. In this case, there is a bean that implements the routing behavior. As discussed this represents a dead end for the crawler that cannot be resolved. It would be necessary to parse the code of the bean class or extend the JMX support. So the crawling result includes one proxy endpoint and the recipient list gateway (cf. Figure 3.4).

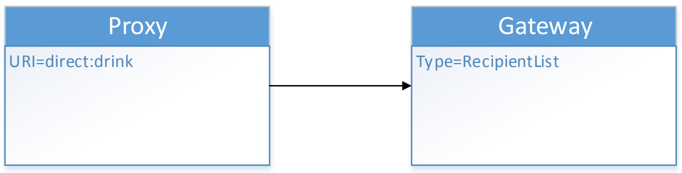


Figure 3.4: The recipient list with the unknown target endpoint.

The previous crawled recipient list gateway invokes either the route with the proxy *seda:coldDrink* or *seda:hotDrink*. The crawled entities are identical besides of the drink method and number of concurrent customers. They represent a pipes and filter pattern. The filter is implemented as Java bean which will be invoked in sequence. First the proxy, then the drink preparation Endpoint followed by the connection endpoint to the next route.

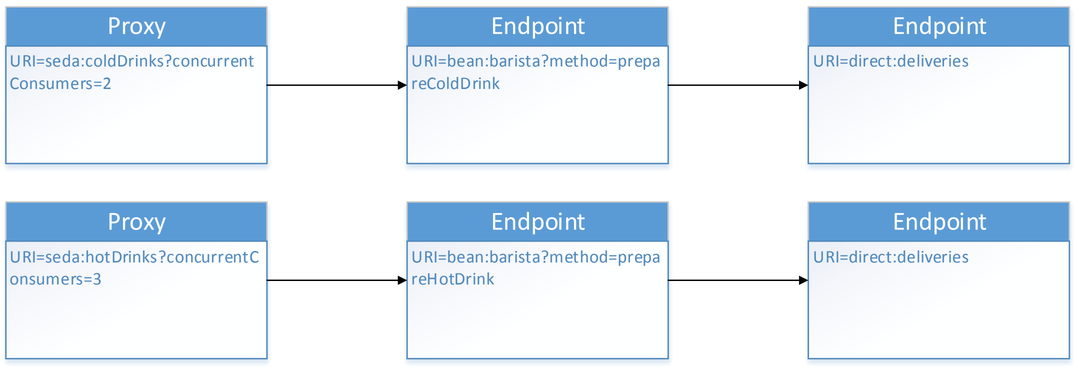

Proxy
URI=seda:coldDrinks?concurrentConsumers=2
Endpoint
URI=bean:barista?method=prepareColdDrink
Endpoint
URI=direct:deliveries
Proxy
URI=seda:hotDrinks?concurrentConsumers=3
Endpoint
URI=bean:barista?method=prepareHotDrink
Endpoint
URI=direct:deliveries

Figure 3.5: Preparation routes of either cold or hot drinks.

This is also a good example that the crawler should not interpret the crawling result. The crawler could judge the prepare Endpoint as a dead end. Because he could determine if this is an entity or a proxy endpoint of another route. It is of advantage to avoid this systematical complexity. The user of the ETG Framework could define a query reassembling the complex structure of the mediator flow. But the query needs some context knowledge that must include insight of the architecture and constitution of the mediator flow. Figure 3.5 illustrates the crawled result of both routes.

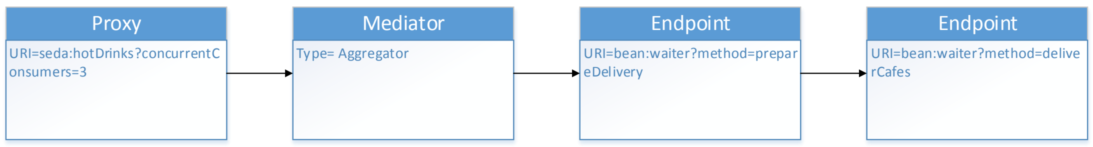

Proxy
URI=seda:hotDrinks?concurrentConsumers=3
Mediator
Type= Aggregator
Endpoint
URI=bean:waiter?method=prepareDelivery
Endpoint
URI=bean:waiter?method=deliverCafes

Figure 3.6: The expected aggregator route elements.

Finally all the prepared items will be aggregated within the final route. Figure 3.6 represents the structure of one proxy, one mediator and two endpoints. Again the endpoints represents a pipes and filter modeling concept.

3.7 Applied Gathering Methods

The user can define Apache Camel routes using a Java DSL in Java classes that can access the *RouteBuilder* entity. The routes can be placed in any class in the ESB project of the enterprise.

Listing 3.9 Example of an camel route configuration using Java DSL

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      errorHandler(deadLetterChannel("mock:error"));
      from("direct:a").to("direct:b");
   }
};
```

There is no central registry that could be accessed to determine the static configuration of the routes.

The route in Listing 3.9 illustrates a programmatically defined route added to the *camel context*. Apache Camel even encourage developers using this concept to setup an Apache Camel mediator flow. This would provide maximum IDE completion and functionality. Furthermore it should also be the most expressive way to define routes in Apache Camel [The13].
The implementation of the scenarios used in this thesis, mainly configures the routing rules using the Apache Camel XML language. The usage of the XML language can be compared to the Spring application context. Both routing description alternatives must be crawled and detected for the static routing information of the Apache Synapse routes. So it is not sufficient to analyze the *camel-route.xml* or the source code.
However Apache Camel provides a sophisticated Java JMX support, the user can monitor and control the ESB with a JMX client. The Camel JMX catalog provides different information sections:

- **Components**
  The components represent a used and supported technology by Apache Camel. For instance the direct- (cf. Section 3.5.2) or the SEDA-component (cf. Section 3.5.5).
- **Context**
  If there is more than one context active in the Camel instance it could be determined in this section.
- **Endpoints**
  Every Camel endpoint container implements the *ManagedEndpoint* interface. This makes them manageable using JMX.
- **Processors**
  All entities that implement the processor pattern are built to consume and send messages. This can be either channels or mediators. Each of these entries are correlated to a route.
- **Routes**
  The Route defines the address that will be exposed for the route invocation. This can also be modeled as an proxy endpoint.

The route information contains all properties in their description formatted as string. Unfortunately these information is not accessible as JMX Attributes. Basically the string represents the structure using a recursive language with the syntax *Object[Object next Object]*. The Object is an internal Camel Object. There is no documentation of the used structure of the description.

Section 3.2 delivers a solution of this issue. The illustrated mapping of the JMX description with the correspondent EAI pattern made a breakdown of the structure possible. This universal syntax and correlation of the JMX description string has also been re-engineered in this Section.
With this solution the method can be used to gather all routing information either defined as Java code or with the Apache Camel XML language. The JMX method also enables the access to statistical information from Apache Camel. Needless to say, because of the sophisticated JMX support there is no other custom management API available.

One should also consider that log files are no useful alternative because the default logging level does not display any routing information. This configuration of the logger can be changed using the *camel context*. It would require to enable *DEBUG level*. Yet the debug output of Apache Camel is verbose and would need additional parsing effort. Hence, the JMX solution can be considered as the superior crawling practice. The JMX approach has been mentioned in the documentation as the recommended service interface for third-party programs.

3.8 Crawler Implementation Details

3.8.1 Crawler Data Model

The crawler model illustrated in Figure 3.7 reflects the JMX entities used by Apache Camel. There are three important entities for the gathering of static and statistical information (i) endpoint, (ii) processor and (iii) route.

The JMX output contains attributes that are needed as information for the ETG Framework plugin. They are modeled as private members of each data model class. The *get*-methods are private because the JMX client logic is integrated in the constructor. The constructor arguments of the CamelEndpoint includes the *MBeanServerConnection*, *ObjectInstance* and the *MBeanInfo* object. The connection is stored in every object to clearly define the instance reference. It is also needed to query attributes from the JMX interface.
Some information can be directly retrieved from the *ObjectInstance*. It encapsulates all information about the source object. The *MBeanInfo* describes the management interface that is exposed by an *MBean*. For example the MBean class *managedSendProcessors* provides a destination property contrary to the *managedProcessor*. Subsequently each object will be instantiated with the same arguments and the information will be gathered automatically inside of the class. The advantage is that the information gathering logic is attached to the object to avoid assignment errors with *set*-methods. This architecture also ensures clear enclosure of the source code and easy code maintenance.

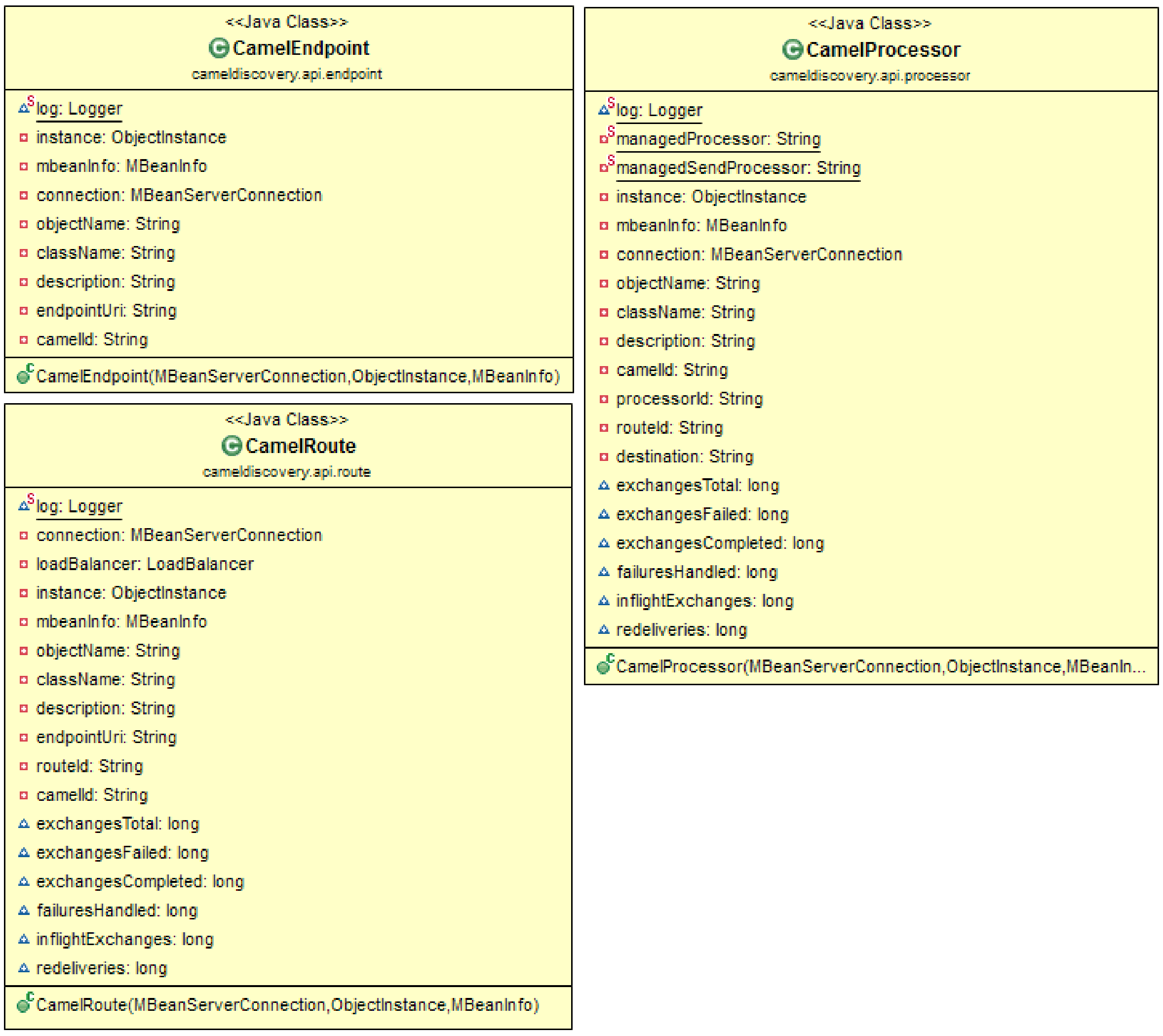


Figure 3.7: The camel crawler data model depicted as UML class diagram

3.8.2 JMX Object Helper

Listing 3.10 MObjectHelper: Query an endpoint using the *getObjectInstance* method

```
public static ObjectInstance getEndpoint(MBeanServerConnection conn, String
    endpointName) throws Exception {
      return getObjectInstance(conn, "endpoints", "\"" + endpointName +
          "\"");
}
```

The Camel crawler project supports a static *MObjectHelper* utility class. Queries on the JMX interface can be very complex and error prone. Interface query functions help the developer to avoid errors and duplicated code. An endpoint can easily be found with the helper function in Listing 3.10 The implementation searches for a given endpoint name in the endpoint category of the Apache Camel JMX interface.

Listing 3.11 MObjectHelper: Query a arbitrarily object instance in JMX

```
private static ObjectInstance getObjectInstance(MBeanServerConnection conn,
    String type, String name) throws MalformedObjectNameException,
    NullPointerException, IOException {
      Set<ObjectInstance> beans = conn.queryMBeans(new
          ObjectName("org.apache.camel:type=" + type + ",name=" + name +
          ",*"), null);
      return beans.isEmpty() ? null : beans.iterator().next();
}
```

The function facilitates the utility function *getObjectInstance* represented in Listing 3.11. The code iterates through all objects with a user defined type. In our case the function returns the appropriate endpoint instance. These are just snippets of one use case of the *MObjectHelper*. The Java class provides 17 custom static functions which enables easy query mechanism using JMX. It also offers simple extension of the camel crawler, if there are more information needed in the future.

3.8.3 Camel Structure Builder

The *CamelEndpoint*, *CamelProcessor* and *CamelRoute* objects (displayed in Figure 3.7) will be added to the *CamelStructure* object illustrated in Figure 3.8 that represents the complete Camel topology.

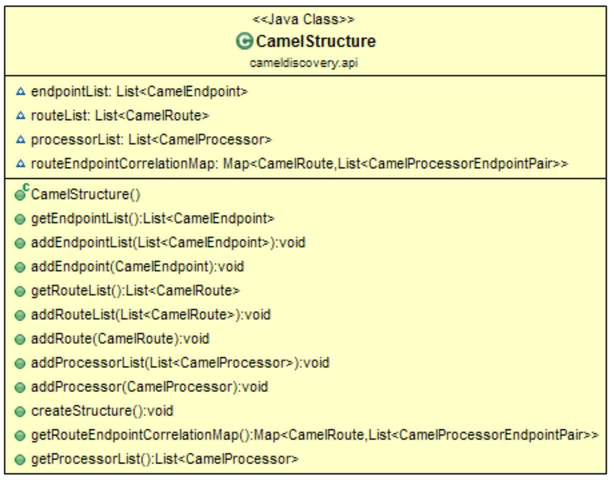

<<Java Class>>
CamelStructure
cameldiscovery.api
endpointList: List<CamelEndpoint>
routeList: List<CamelRoute>
processorList: List<CamelProcessor>
routeEndpointCorrelationMap: Map<CamelRoute,List<CamelProcessorEndpointPair>>
CamelStructure()
getEndpointList():List<CamelEndpoint>
addEndpointList(List<CamelEndpoint>):void
addEndpoint(CamelEndpoint):void
getRouteList():List<CamelRoute>
addRouteList(List<CamelRoute>):void
addRoute(CamelRoute):void
addProcessorList(List<CamelProcessor>):void
addProcessor(CamelProcessor):void
createStructure():void
getRouteEndpointCorrelationMap():Map<CamelRoute,List<CamelProcessorEndpointPair>>
getProcessorList():List<CamelProcessor>

Figure 3.8: The Apache Camel internal crawler structure

The crawler iterates over the Apache Camel JMX management interface and then adds all endpoints, routes and processors to the structure. After this is done, all objects can be correlated together. That makes it easier because the user of the Camel crawler does not need any Apache Camel specific knowledge to correlate the JMX entities together. The *createStructure()* function starts the clustering algorithm. The algorithm in Listing 3.12 is performing several clustering steps.

Listing 3.12 Simplified Apache Camel structure clustering algorithm

```
for (CamelProcessor processor : processorList) {
      for(CamelRoute route : routeList) {
            if(route.getRouteId().equals(processor.getRouteId())) {
                  if(!routeEndpointCorrelationMap.containsKey(route)) {
                  *** add new processor and endpoint pair to route ***
                  }
                  for(CamelEndpoint endpoint : endpointList) {
                  if(endpoint.getEndpointUri().equals(processor.getDestination()))
                      {
                  *** Processor is a gateway. Add more Endpoints to the processor
                      subList ***
                        }
                  }
            }
      }
}
```

The idea behind this algorithm is that every route has mediators that connect one or more endpoints. Each processor represents a mediator that connects an endpoint. This connection can be resolved by mapping the endpoints to the processor endpoint address. The clustering creates a simple data-structure that can be described as follows:
"a Apache Camel route contains one or more processors with zero or more endpoints"
This is the underlying clustering idea behind this algorithm. An external application like the ETG Framework plugin can simply iterate through the routes and can access all mediator, gateway, condition and endpoint entities that are deployed in the Apache Camel instance.

3.8.4 Camel JMX Connection Client

The crawler implements a custom JMX client to establish the connection to the Apache Camel management. The documentation is inaccurate regarding the correct connection string that a client needs to connect a Camel instance. There are several possible connection strings, depending on the implementation and used Apache Camel version. The default connection string is *URL:service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi*. The Crawler API provides a connection tester that will try several connection methods. The default configuration of Camel enables the JMX support.
The *JMXConnector* class uses the *javax.managment.remote.JMXConnectorFactory* to establish

the connection. The *MBeanServerConnection* provides access to all JMX services that Apache Camel provides.

3.9 Discussion of Crawling Results

The crawling results represent a direct view on the Camel configuration. As mentioned in Chapter 3.6.2, there are several limitations in that discovery.

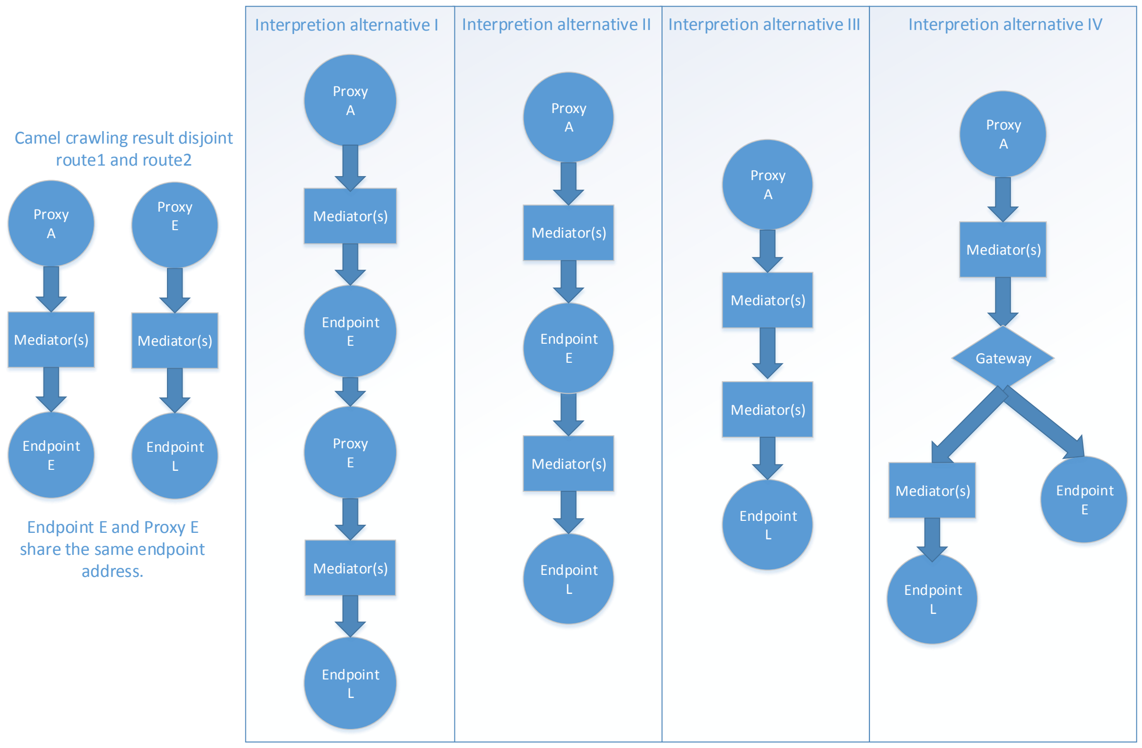


Figure 3.9: Example of several interpretation alternatives of two Apache Camel routes

- **Correct distinction between configuration and crawling results**
  The Camel crawler only detects patterns that are listed in Section 3.3.1. Channels are also part of the internal camel routing process. The crawler abstracts this concept and implicitly represents a channel as a relation between processors and endpoints.
- **Routing logic implemented as Java beans**
  The Apache Camel crawler has no insight on routing logic inside of Java beans. The coffee shop example in Section 3.6.2 illustrated this limitation in practice. The routing algorithm behind the bean is out of scope of the crawler. This leads to dead ends in the camel route chain.

- **Concatenation of routes**
  Camel routes typically represent a complete mediator flow. Every route defines one activity or functionality of that message flow. The crawler does not join possible matching routes because it could lead to wrong representation of the structure. The example in Figure 3.9 explains the difficulties of reassembling routes:
  The Camel developer defines two different routes in the Apache Camel ESB:
  Route A can be invoked with the proxy endpoint P listening to the addressee *direct:A*. After some processing steps the endpoint E with the addressee *direct:E* receives the message. Route B can be invoked with the same addressee as endpoint E and acts in this route like a proxy endpoint. After some processing the request will be handled to the last endpoint L.
  In a nutshell the Endpoint E and Proxy E are the same endpoint exposing the same address. This can lead to multiple interpretation alternatives how both routes are correlated together. Four possible interpretation alternatives are illustrated in Figure 3.9.
  On the other hand Alternative I assume that the incoming request logic is separated from the outgoing logic. Apache Synapse sequences often deal with this separated endpoint implementation model. Alternative II joins the endpoint E with the proxy E. Both entities would act as one entity. But E can only be invoked with proxy A. Alternative III removes endpoint E, because it could only implement internal message transport mechanics. Finally the Alternative IV adds a gateway to connect both routes. Because it depends on the payload of the request if the Endpoint E is the last processing step or could trigger additional mediator flow.
  The untouched disjoint representation does not make any assumptions of the correlation between both routes. This can be considered as a real snapshot of the Camel structure. The ETG Framework provides graph transformation that perform an automatically interpretation of the Camel structure. This transformation cannot be done without information about the business context of the routes and endpoints.

CHAPTER 4

FOCUS ON APACHE SYNAPSE

This Chapter deals with Apache Synapse describing the general analysis and research to achieve a sophisticated information crawler implementation. Furthermore, Section 4.1 classifies Apache Synapse using characteristics from David Chappell. Followed by a summary about the architecture and concepts of Apache Camel in Section 4.2, using the chronological order of all involved Apache Camel parts in the message flow. The mediator flow can be defined using the functional components discussed in Section 4.3. These components has been mapped in Section 4.4 to the Enterprise Integration Patterns. The concepts of Apache Synapse has been adapted by WSO2 ESB introduced as a user story in Section 4.5. Finally all the research has been implemented as Apache Synapse information crawler and evaluated using custom Apache Synapse scenario deployments explained in Section 4.6. The technical details are discussed in Section 4.7 and 4.8 and the results will be discussed in Section 4.9.

4.1 Classification

Apache Synapse can be described as a lightweight and high-performance Enterprise Service Bus. It provides support for Web services and REST. The latest stable release of Apache Synapse is version 2.1 from january 2012. The Synapse architecture provides fundamental services for complex architectures like message routing and message transformation [Thee]. Table 4.1 compares the ESB characteristics defined by D. Chappell [Cha09] with the Apache Synapse functionality. Apache Synapse meets the requirements discussed in Chapter 2.2 formalized as characteristics of an Enterprise Service Bus [Cha09].

ESB Characteristics	Synapse functionality
Pervasiveness	Every Application can be anonymously added to the configuration of Apache Synapse. Typically the endpoint will be discovered using a service address or a WSDL file. Apache Synapse just need the address of the endpoint that is being used to invoke the message exchange with the supported transportation protocol.
Standards-Based Integration	Synapse supports Web services and REST. The Internet-based standards SOAP and XML are supported as content interchange formats. It provides also support for several industry standard protocols like HTTP/S, Mail, Java Message Service, TCP, UDP, Virtual File System [a], Short Message Service [b], XMPP[c] and FIX [d].
Highly distributed, event-driven SOA	Synapse can invoke any application that is accessible via a supported transport protocol. Synapse can be used as an event source and enables integration of other systems based on an Event Driven Architecture (EDA).
Security and reliability	WS-Security is one example of a supported security feature in Apache Synapse. ASF Project Security Information Homepage gives the user insight about current security issues and their solutions. The reliability of the provided Web services are ensured with the support of WS-Reliable Messaging.
Orchestration and process flow	Orchestration is implemented as a routing engine in the core of the ESB. Apache Synapse provides a sophisticated catalog of mediators.

Table 4.1: Comparison of ESB characteristics and Apache Synapse functionality

[a]VFS - http://commons.apache.org/proper/commons-vfs/
[b]GSM Technical Specification - http://www.etsi.org/
[c]XMPP Protocols - http://xmpp.org/xmpp-protocols/
[d]FIX Standard - http://www.fixtradingcommunity.org/

4.2 Architecture and Concept

Figure 4.1 provides an overview of the Apache Synapse high level architecture. It shows the interactions between the main components. The architectural concept is driven by the messaging handling of the software. Each component will be discussed in order of the components work with the message flow, beginning with the receiving of an message:

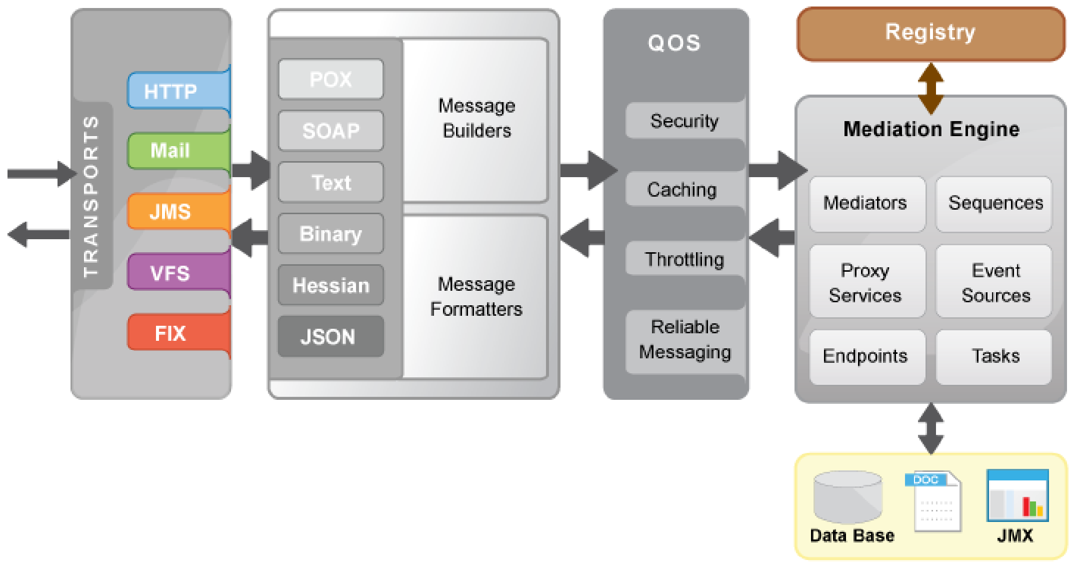


Figure 4.1: Overview of the main components of the Apache Synapse architecture [Thed]

An application sends a message to the Apache Synapse ESB and it will be proceeded by the transport component. The system picks an implementation of a transport that is capable of receiving and sending messages.
Synapse only processes the content as needed. This model is implemented in the message formatters and builder. They create objects that are handled as internal representation of the request. The message builder identifies the message data type and assembles the Synapse XML format. The message formatters are doing the reverse work while converting the messages back to the original format [Fre].
After that the message will be handled by Quality of Service (QoS) components that provide security, caching, throttling and reliable messaging features.
Now the mediation engine of Apache Synapse handles the request. The routing logic is defined using sequences. The sequences contain a routing chain with proxy services, mediators and endpoints. Theses features are discussed in Section 4.3. The Mediator Engine can also access a local or remote registry that provides additional configurations. There is also access to external databases that can be used.
Finally after the request propagation to the endpoint, the generated reply takes the reverse way through the architecture. Again the QoS components will be applied to the reply and the message formatted will transform the reply to a compatible format for the calling application. In a final step suited transport component will deliver the reply message.

Listing 4.1 The syntax outline of a sequence definition

```
<sequence name="string" [onError="string"] [key="string"] [trace="enable"]
    [statistics="enable"]>
  mediator*
</sequence>
```

Listing 4.2 Example of an main Synapse sequence with three mediators

```
<sequence name="main" onError="errorSequence">
   <log/>
   <property name="test" value="test value"/>
   <send/>
</sequence>
```

4.3 Functional Components

The Mediator Engine can be deployed using a XML-based configuration language. The SOA components can be used an orchestrated using a large collection of mediators that are able to solve complex enterprise integration scenarios. This Section introduces the concepts that are build with sequences, mediators and endpoints [Thee].
The outlines are using POSIX Extended Regular Expression (ERE) syntax elements. The + sign defining (0,1) occurrence and * sign defining (0,n) occurrence.

4.3.1 Sequence

A sequence can also be described as mediator sequence that is basically a list of mediators. Listing 4.1 displays the syntax of a sequence element that only allows to define a list of mediator elements. The sequence has to be named uniquely. The Synapse configuration contains two special sequences with reserved names:

- **Main**
  The main-sequence will handle all requests that are accepted for the message mediation. It acts like a default message route.
- **Fault**
  The fault-sequence is handling every error encounter during processing of an message.

Even if the user does not declare these sequence in the configuration, the ESB will automatically generate suitable main- and fault-sequences.

This chain of mediators can be compared with the pipe and filter pattern. The messages are sent through a pipe of filters that are represented by the mediators. For instance the filter mediators can transform messages to interact services without changing their implementation. There are similarities with the previous discussed Apache Camel routes. But in contrast to Camel the routes are not nested and typically execute a complete task. Usually in Apache Camel it is best practice to chain the routes instead of the mediators. Finally an endpoint will

be invoked. An example of Synapse sequence containing three mediators is illustrated in Listing 4.2.
But it is also possible to chain sequences like in Apache Camel. This can be useful to reuse the different processing tasks in the Synapse configuration. This usage can be compared with the functionality of a procedure in a programming language. The code of a procedure represents a functionality that can be invoked in any other context. The sequence example in the Listing 4.2 for instance reuses the *errorSequence* to handle a possible exception in a mediator. Figure 4.2 illustrates a possible Apache Synapse routing. The proxy endpoints receive and deliver the messages to the client.

4.3.2 Proxy

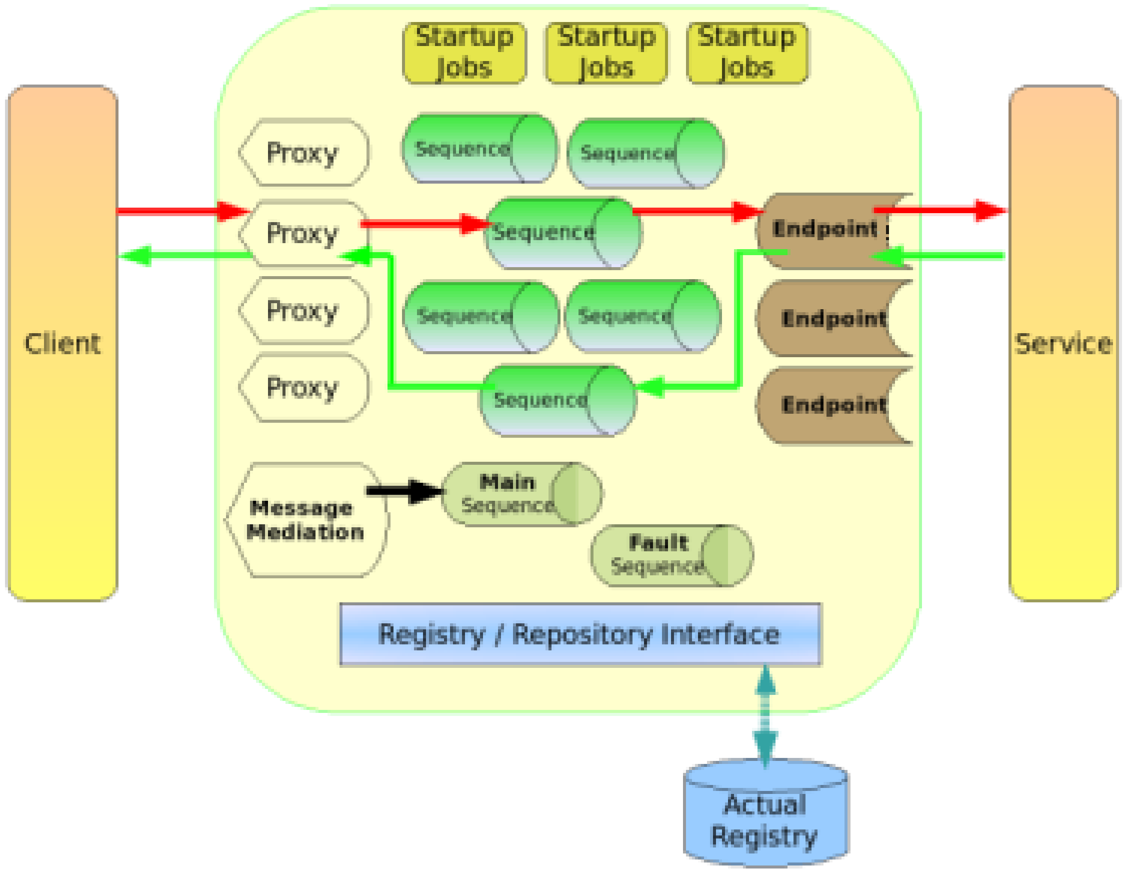


Figure 4.2: The Synapse messaging flow structure [Thed]

Typically Synapse ESB mediates between the calling application and the internal backend service. The mediation is orchestrating the message flow and invokes the mediator that perform authentication, validation, transformation, routing based on the content etc. and then executes the destination target endpoint. This functionality is defined in the Apache Synapse sequence.

Normally it is not possible, except for the main route, that a route can get a message without a proxy endpoint. The proxy service exposes the specific transports through an underlying Apache Axis2 server. Each of these services could define a sequence or a direct endpoint. Basically its the communication layer between the client and the internal Synapse endpoint service.

Listing 4.3 The syntax outline of an Apache Synapse proxy service

```
<proxy name="string" [transports="(http |https |jms |.. )+|all"]
    [pinnedServers="(serverName )+"] [serviceGroup="string"]>
  <description>...</description>?
  <target [inSequence="name"] [outSequence="name"] [faultSequence="name"]
      [endpoint="name"]>
    <inSequence>...</inSequence>?
    <outSequence>...</outSequence>?
    <faultSequence>...</faultSequence>?
    <endpoint>...</endpoint>?
  </target>?
  <publishWSDL key="string" uri="string">
    ( <wsdl:definition>...</wsdl:definition> |
        <wsdl20:description>...</wsdl20:description> )?
    <resource location="..." key="..."/>*
  </publishWSDL>?
  <enableAddressing/>?
  <enableSec/>?
  <enableRM/>?
  <policy key="string" [type="(in | out)"]/>? // optional service or message
      level policies such as (e.g. WS-Security and/or WS-RM policies)
  <parameter name="string">      // optional service parameters such as (e.g.
      transport.jms.ConnectionFactory)
    string | xml
  </parameter>
</proxy>
```

Listing 4.3 shows the complexity of a proxy and the nested children. One the one hand the *inSequence* mediator describes the mediator flow for the incoming requests. Mediators on that incoming path will be annotated with *(In path)*. On the other hand the *outSequence* describing the mediator flow of the response path. These Mediators will be annotated with *(Out path)*.
As mentioned the proxy can also directly target an endpoint. The service is designed to publish a WSDL file for the client as service description.

4.3.3 Endpoint

Endpoints are backend services that can be defined in the Synapse configuration. Service endpoints can be either address endpoints or WSDL endpoints. They are both addressed with a URI. But the WSDL endpoint could also have an inline definition within the configuration. There are also internal endpoints like the failover endpoint that is a backup for the first listed endpoint in the published WSDL configuration.
Apache Synapse also defines the loadbalancer as an endpoint that invokes backend endpoints

depending on the load algorithm. This semantic is different from Apache Camel. Similarly, the recipient list endpoint can be defined using the recipient list element. The structure is similar to a loadbalance endpoint, with the difference that the message will be send to a list of recipients. This implements the *recipient list* pattern.

4.3.4 Mediator

Mediators can perform a specific function such as sending or filtering messages. Apache Synapse provides a mediator library that provides functionality for implementing the Enterprise Integration Patterns. The functionality of each available mediator will be discussed the subsequent sections. These filters are processing the messages in different ways. It is necessary to illustrate those concepts. They can affect the message flow and final routing to an endpoint [WSO13].
Section 4.3.5 discusses the core mediators representing basic functional expressions. Followed by the section 4.3.6 with filter type mediators. Specialized mediators for SOAP transformation are covered in section 4.3.7. Followed by the Extension Mediators in section 4.3.8 and finally the advanced mediators are discussed in 4.3.9.

4.3.5 Core Mediators

The Core mediators represent the basic functional expressions to create a message flow with Apache Synapse. They are very useful in variety of scenarios and they are utilized to improve the messages handling. There are four different utility mediators:

Drop Mediator:

```
<drop/>
```

The Drop-Mediator will drop the current messages from the flow. The message flow will be terminated.

Log Mediator:

```
<log [level="simple|full|headers|custom"] [separator="string"]
               [category="INFO|DEBUG|WARN|ERROR|TRACE|FATAL"]>
   <property name="string" (value="literal" | expression="xpath")/>*
</log>
```

The mediated messages can be logged at any given position in the message flow. This log will go into the standard Apache Synapse log files. This is a log4j[1] logging mechanism and can be configured further outside of the Synapse configuration file. The logging detail, category and

[1]http://logging.apache.org/log4j/2.x/

property name can be specified. This mediator will be used to establish routing statistics in the ESB.

Property Mediator:

```
<property name="string" [action=set|remove] [type="string"] (value="literal" |
    expression="xpath") [scope=default|transport|axis2|axis2-client]
    [pattern="regex" [group="integer"]]>
   <xml-element/>?
</property>
```

This property can manipulate some properties of the messages. This can be very helpful for controlling the runtime behavior of a message. Important Message detail could be added and changed while processing. A possible use case could be a mediator that stores a intermediate results for further processing by other mediators.

Send Mediator:

```
<send [receive="string"]>
   (endpointref | endpoint)?
</send>
```

This send operation can transport requests to endpoints. But this could be also be used to create response messages back to the client. A send mediator without configuration of any child endpoints, will forward the messages to an implicit endpoint (e.g. inspecting the "to" header of the message). If it is a response message of a backend client then the message will be send to the requester client.

4.3.6 Filter Mediators

Simple Filter Mediator:

```
<filter (source="xpath" regex="string") | xpath="xpath">
   mediator+
</filter>
```

The filter validates messages and passes them to the next one if the regular expression is matched. It is possible to use if/else semantic for different evaluations that would make conditional actions necessary. The evaluation scenario in Section 4.6.1 demonstrates such a usage as content based router.

In/Out Mediator:

```
<in>
   mediator+
</in>

<out>
   mediator+
</out>
```

The in mediator only applies to incoming messages flow and the out mediator to out-going message flow. According to the origin of the message (from the client or from the backend endpoint) different mediators will be applied.

Switch Mediator:

```
<switch source="xpath">
   <case regex="string">
      mediator+
   </case>+
   <default>
      mediator+
   </default>?
</switch>
```

This mediator acts like a switch statement in Java. The source XPath will be executed on the message and the resulting value will be matched against the case expression.

4.3.7 Transformation Mediators

The transformation mediators are specialized for SOAP messages. They are specialized on different parts and transformation of a message.

Header Mediator:

```
<header name="qname" (value="literal" | expression="xpath") [action="set"]/>
<header name="qname" action="remove"/>
```

Header mediator can change the header from the current SOAP specification. The name attribute describes the field which can be changed in the header.

MakeFault Mediator:

```
<makefault [version="soap11|soap12|pox"] [response="true|false"]>
   <code (value="literal" | expression="xpath")/>
   <reason (value="literal" | expression="xpath")/>
   <node>...</node>?
   <role>...</role>?
  (<detail expression="xpath"/> | <detail>...</detail>)?
</makefault>
```

This transformations mediator can for instance transform a SOAP message into a fault message. The error handling must be invoked after that.

Payload Factory Mediator:

```
<payloadFactory>
   <format>"xmlstring"</format>
   <args>
      <arg (value="literal" | expression="xpath")/>*
   </args>
</payloadFactory>
```

It creates a new payload for the SOAP message. This is provided by using a XPath expression against the existing Payload of the SOAP message or message context. But it is also possible to change the payload with a static value.

URL Rewrite Mediator:

```
<rewrite [inProperty="string"] [outProperty="string"]>
   <rewriterule>
      <condition>
      ...
      </condition>?
      <action [type="append|prepend|replace|remove|set"] [value="string"]
        [xpath="xpath"] [fragment="protocol|host|port|path|query|ref|user|full"]
            [regex="regex"]>+
   </rewriterule>+
</rewrite>
```

Every URL values in the message can be modified and transformed. By default the to header of the message will be changed by the rewrite rules.

XQuery Mediator:

```
<xquery key="string" [target="xpath"]>
   <variable name="string" type="string" [key="string"] [expression="xpath"]
       [value="string"]/>?
</xquery>
```

In this mediator the key attribute targets a specific XQuery transformation, the optimal target attribute specifies the part of the message that should be transformed. The key with the XPath expression selects the variable.

XSLT Mediator:

```
<xslt key="string" [source="xpath"] [target="string"]>
   <property name="string" (value="literal" | expression="xpath")/>*
   <feature name="string" value="true | false" />*
   <attribute name="string" value="string" />*
   <resource location="..." key="..."/>*
</xslt>
```

XSLT Mediator transforms a selected element of a SOAP message payload.

4.3.8 Extension Mediators

It is possible to create custom mediators for Apache Synapse. These extension mediators assist the user with the creation of an instance. Synapse provides different interfaces for several implementation alternatives:

- **Class Mediator**
  By implementing the *org.apache.synapse.api.Mediator* interface. The user can integrate custom Java mediators.

- **POJO Command Mediator** POJO is a popular Command design pattern to encapsulate method calls to invoke an object. Implementing the *org.apache.synapse.Command* interface with an execute() signature provides the functionality of the mediator.
- **Script Mediator**
  Variety of script languages like JavaScript, Python and Ruby can be used to implement custom mediators.
- **Spring Mediator**
  Like the Class Mediator it is possible to invoke and instantiate Spring beans.

4.3.9 Advanced Mediators

Aggregate Mediator:

```
<aggregate [id="string"]>
   <correlateOn expression="xpath"/>?
   <completeCondition [timeout="time-in-seconds"]>
      <messageCount min="int-min" max="int-max"/>?
   </completeCondition>?
   <onComplete expression="xpath" [sequence="sequence-ref"]>
      (mediator +)?
   </onComplete>
</aggregate>
```

This acts like the Message Aggregation Pattern described in the Enterprise Integration Patterns combining messages together. They have to *correlateOn* a XPath expression and will be collected till the completion condition is met. If the *completionCondtion* was met the messages will be merged and forwarded to the *onComplete* sequence.

Cache Mediator:

```
<cache [id="string"] [hashGenerator="class"] [timeout="seconds"] [scope=(per-host
    | per-mediator)]
      collector=(true | false) [maxMessageSize="in-bytes"]>
   <onCacheHit [sequence="key"]>
      (mediator)+
   </onCacheHit>?
   <implementation type=(memory | disk) maxSize="int"/>
</cache>
```

This mediator detects already processed messages. If an incoming message has already been sent and can be correlated in the cache then the *onCacheHit* sequence will be invoked.

Callout Mediator:

```
<callout serviceURL="string" [action="string"]>
   <configuration [axis2xml="string"] [repository="string"]/>?
   <source xpath="expression" | key="string">
   <target xpath="expression" | key="string"/>
</callout>
```

This is basically a blocking call on an external service. The response will be attached to the current message context as a property. It cannot be used with HTTP/s protocol (because of the blocking characteristics).

Clone Mediator:

```
<clone [id="string"] [sequential=(true | false)] [continueParent=(true | false)]>
   <target [to="uri"] [soapAction="qname"] [sequence="sequence_ref"]
       [endpoint="endpoint_ref"]>
      <sequence>
         (mediator)+
      </sequence>?
      <endpoint>
         endpoint
      </endpoint>?
   </target>+
</clone>
```

The incoming messages will be copied several times. These clones are identical copies of the incoming messages and can be processed in parallel or sequential.

DBLookup Mediator:

```
<dblookup>
      ...
</dblookup>
```

The DB Mediator can process SQL Statements on a defined SQL Database. The resulting data will be stored in the Synapse message context. Similar to this behavior is the DBReport mediator. Instead of reading data it will write data to a given database.

Iterate Mediator:

```
<iterate [id="string"] [continueParent=(true | false)] [preservePayload=(true |
    false)] [sequential=(true | false)]
      (attachPath="xpath")? expression="xpath">
   <target [to="uri"] [soapAction="qname"] [sequence="sequence_ref"]
       [endpoint="endpoint_ref"]>
      <sequence>?
      </endpoint>?
   </target>+
 </iterate>
```

The Iterator splits the message in multiple items using an XPath expression. For each item a new message will be created forwarded to a sequence or an endpoint.

RMSequence Mediator:

```
<RMSequence (correlation="xpath" [last-message="xpath"]) | single="true"
    [version="1.0|1.1"]/>
```

Creating a sequence of messages to communicate via WS-Reliable Messaging.

Store Mediator:

```
<store messageStore="string" [sequence="sequence-ref"]>
```

The messages can be harvested in a message storage.

Throttle Mediator:

```
<throttle [onReject="string"] [onAccept="string"] id="string">
   (<policy key="string"/> | <policy>..</policy>)
   <onReject>..</onReject>?
   <onAccept>..</onAccept>?
</throttle>
```

This can be used to control the load and limiting as well as the concurrency of messages. This is defined in WS-Policy. Depending on the load the *onReject* or *onAccept* sequence will be used.

Transaction Mediator:

```
<transaction
    action="new|use-existing-or-new|fault-if-no-tx|commit|rollback|suspend|resume"/>
```

Transaction Mediator enables transaction procession of defined child mediators.

4.4 Pattern Overview

The functional components discussed in Section 4.3 can be mapped to Enterprise Service Patterns (EIP) [HW02]. This proves that it is possible to integrate applications using EIP with Apache Synapse. All the necessary patterns are present in the standard mediator library. These patterns are the basic for the evaluation of the Apache Synapse crawler. Also the generic data model can be applied for the transformation because its complete with EIP. This correlation is needed because the Apache Synapse documentation does not provide a complete mapping. The Table 4.2 illustrates the EIP with the corresponding Apache Synapse concept.

Messaging Systems	
Message Router	Simple filter mediator with conditions.
Message Translator	Transformation mediator can translate a message to an arbitrary format.
Message Endpoint	Provided by Apache Synapse as endpoint with same semantics.
Message Routing	
Content-Based Router	Simple filter mediator with conditions.
Message Filter	Filter mediators.
Dynamic Router	Not supported. But the concept can be implemented using an extension mediators. For instance, the class mediator can implement state-full routing.
Recipient List	A recipient list endpoint can be used to send a single message to a list of recipients
Splitter	The iterator mediator with the clone mediator (cf. example usage in 4.6.2)
Aggregator	Aggregator mediator
Resequencer	Not supported. But the concept can be implemented using an extension mediators.
Composed Message	splitter mediator followed by router mediator that connects a gateway. The gateway have several elements as children for the sub-messages. These elements all have the aggregator as child.
Scatter-Gather	Implemented in section 4.6.2 combination of splitter and aggregator
System Management	
Control Bus	Simple filter mediator targeting sequence with additional mediators
Detour	Simple filter mediator targeting sequence with additional mediators
Wire Tap	Content based router with recipient list
Message History	Log mediator
Message Store	Store mediator
Smart Proxy	mediator with id property for the request and response message path

Table 4.2: Correlation of Synapse mediators and Enterprise Integration Patterns

4.5 User Stories

WSO2 Enterprise Service Bus is built on the Apache Synapse project. It uses the exactly same core architecture. The WSO2 ESB has enhanced features like a management console that assists the configuration of mediator flows with a web interface. The ESB can be installed on local servers, private clouds or in an Infrastructure as a Service cloud like Amazon EC2. But WSO2 also provides a public cloud that allows on-premise out of the box functionality. Like in Apache Synapse, the configuration is provided by XML files using the Synapse XML schema definitions [WSO13].
There are several released business case studies that uses the WSO2 ESB product:
EBay considered a new internal system or adopted a third-party technology. EBay employed deep analysis of each ESB products on the market and finally has chosen WSO2 ESB with a 24 hour support contract. The product suits the requirements of the online marketplace in both speed and reliability. After one year the bus handles over 1 billion calls per day. The business functions are supported with the routing, orchestration and service chaining features of WSO2. According to the case study the instances of the ESB remained fast, stable and ensured high availability with very efficient resource utilization [WSO12a].
Another use case is the Alfo-Bank that is part of the international Alfa Group Consortium. The bank used a single handled banking system that was integrated as a classical enterprise system. The structure involved large amounts of point-to-point connections between systems. The new core banking with WSO2 has to handle the hard-to-change legacy applications that were tightly coupled. These problems were handled by the WSO2 Enterprise Service Bus with effective collaboration, management and mediation of Web services with external legacy applications [WSO12b].

4.6 Evaluation Scenarios

The Apache Synapse packed includes a large catalog of sample configurations. They demonstrate several features of the ESB. The general deployment and installation of the samples is described in Appendix A.
Despite of this large amount of examples, this Section only focuses on the enterprise integration pattern listed in the section 4.4. These patterns are needed to build the business example of a coffee shop (cf. Section 3.6.2) with Apache Synapse.
Those examples are provided in the Apache Synapse source package. They represent developers best practice in the development and usage of the patterns [Thee].

4.6.1 Content Based Routing Example

Listing 4.4 Example: Content based routing with Apache Synapse

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="main">
      <filter source="get-property('To')" regex=".*/StockQuote.*">
         <then>
            <send>
               <endpoint>
                  <address uri="http://../../SimpleStockQuoteService"/>
               </endpoint>
            </send>
            <drop/>
         </then>
      </filter>
      <send/>
   </sequence>
</definitions>
```

Listing 4.4 represents a simple routing example. The filter mediator (cf. Section 4.3.6) can access the message data and validate the content with a regular expression. In this case Apache Synapse accepts all kinds of messages but only route messages from *.*/StockQuote.** to the endpoint.
For instance, the header *http://localhost:8280/services/StockQuote* matches with the regular expression of the child mediator. Finally the sequence will be terminated with the drop mediator [Thef].
The crawler detected four Entities in the following order:

1. *Filter* Mediator with the properties:
   source = get-property('To') regex = *.*/StockQuote.**
2. *Then* Mediator
3. *Send* Mediator
4. Endpoint
   address = `http://localhost:9000/services/SimpleStockQuoteService`
5. *Drop* Mediator

4.6.2 Aggregation and Splitter Example

Listing 4.5 Example: Aggregation and Splitter with Apache Synapse

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="SplitAggregateProxy">
      <target>
         <inSequence>
            <iterate xmlns:m0="http://services.samples"
                expression="//m0:getQuote/m0:request"
                   preservePayload="true" attachPath="//m0:getQuote">
               <target>
                  <sequence>
                     <send>
                        <endpoint>
                           <address uri="http://../../SimpleStockQuoteService"/>
                        </endpoint>
                     </send>
                  </sequence>
               </target>
            </iterate>
         </inSequence>
         <outSequence>
            <aggregate>
               <onComplete xmlns:m0="http://services.samples"
                        expression="//m0:getQuoteResponse">
                  <send/>
               </onComplete>
            </aggregate>
         </outSequence>
      </target>
   </proxy>
</definitions>
```

This scenario shows how aggregation and splitter can be used in the Synapse configuration. The proxy *SplitAggregateProxy* is used to automatically send a number of request containing in one message. Synapse iterates trough each request and sends them as separate requests to the endpoint. After the processing the endpoint returns reply messages that will be aggregated. After all, correlated messages have been collected the response will be sent to the calling application.
The crawler inspected the example in Listing 4.5 and returned following different entities:

1. Proxy Endpoint with the properties:
   name = *SplitAggregateProxy*

2. *Target* Gateway

   (In path) 1. *InSequence* Mediatator

   (In path) 2. *Target* Mediatator

(In path) 3. *Sequence* Mediatator

(In path) 4. *Send* Mediatator

(In path) 5. Endpoint
address = `http://localhost:9000/services/SimpleStockQuoteService`

(Out path) 1. *OutSequence* Mediatator

(Out path) 2. *Aggregate* Mediatator

(Out path) 3. *onComplete* Mediatator

(Out path) 4. *Send* Mediatator

This is a very interesting and complex case and made several issues during development. There are different routing semantics depending on the direction of the message. The *in path* is applied to the request message, instead the *out path* to the reply message. Target handle the incoming and out-going messages. For compatibility reasons with the generic data model (cf. Section 5) the crawler change the target mediator to a gateway.

4.6.3 Load Balance Example

Listing 4.6 Example: LoadBalancer with Apache Synapse

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="main" onError="errorHandler">
      <in>
         <send>
            <endpoint name="dynamicLB">
               <dynamicLoadbalance failover="true"
                              algorithm="org.apache.algorithms.RoundRobin">
                  <membershipHandler
                        class="org.apache.synapse.Axis2LoadBalanceMembershipHandler">
                     <property name="applicationDomain"
                         value="apache.axis2.app.domain"/>
                  </membershipHandler>
               </dynamicLoadbalance>
            </endpoint>
         </send>
         <drop/>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
   <sequence name="errorHandler">
      <makefault response="true">
         <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
             value="tns:Receiver"/>
         <reason value="COULDN’T SEND THE MESSAGE TO THE SERVER."/>
      </makefault>
      <send/>
   </sequence>
</definitions>
```

Load balancing is very useful in an ESB to enable high scalability of the system. It is also possible that the Apache Synapse mediator engine changes the amount of endpoints depending of the dynamic loadbalancer control.
To execute the example in Listing 4.6 it is necessary to run 3 instances of the Axis2 server with different HTTP and HTTPs ports. Please consult the Appendix A for the example deployment. The client is implemented to send 100 request messages to the Synapse instance. They will be distributed among the three endpoints. After that, the message will be sent back with the information which server has processed the message.
The crawler should detect several entities in the following order:

1. *Main* sequence

   (In path) 1. *In* Mediatator

   (In path) 2. *Send* Mediatator

(In path) 3. *Endpoint* Mediatator

(In path) 4. *Loadbalance* Mediatator

(In path) 4.a Endpoint
address = `http://localhost:9001/services/LBService1`

(In path) 4.b Endpoint
adress = `http://localhost:9002/services/LBService2`

(In path) 4.c Endpoint
adress = `http://localhost:9003/services/LBService3`

(Out path) 1. *Out* Mediatator

(Out path) 2. *Send* Mediatator

2. *errorHandler* sequence

   2.1 *Makefault* Mediator

4.7 Information Gathering Methods

Apache Synapse only deployment method is the import of XML files containing routing configurations. All these files are stored in the *repository/conf/synapse-config* directory of the Apache Synapse installation. This file contains all the functional components mentioned in this Section.
It is important to mention that the WSO2 ESB (cf. Section 4.5) also allows to configure with a web plugin. But this is only a creation tool for the Synapse configuration files.
The XML files represent a full snapshot of the Apache Synapse ESB. These configurations can be crawled and analyzed for routing information.
Unfortunately the JMX support of Apache Synapse has not been completely implemented yet. The management interface does not offer any static routing information. In contrast, WSO2 ESB implemented a more sophisticated JMX support, that hopefully will be contributed to the next Apache Synapse release. In the current state Apache Synapse provides no alternatives to get grip on the static routing data.
On the other hand, the dynamical information can be looked up in JMX. Synapse also provides logging mechanism with the log element that is part of the core mediators (cf. section 4.3.5). But this element is optional and the administrators would have to enhance their message flows with the log mediator to enable the feature. But JMX provides statistical information of the Apache Synapse ESB. The dataset only contains information from the running ESB instance. A reset would wipe the data and the crawler could not include the data.
This approach is a trade-off between easy usage for the enterprise IT and the possible amount of data being collected. But the ESB is considered as a long running application. And the log files could be located on an arbitrary place that is not reachable for the crawler. Considering this, the information from JMX are substantial, complete and easy to collect.

4.8 Crawler Implementation Details

4.8.1 JAXB parsing

The Java Architecture for XML Binding (JAXB) is used to convert XML schema documents into Java objects. The content of the objects represent content of the XML document that has been unmarshaled according to the JAXB schema binding. Every object class is using code annotations that describes the mapping between the java and XML. representation of the data. This annotation can be done manually, but JAXB provides a binding compiler that can be launched using the xjc tool. Eclipse also provides a xjc plugin that makes the Java model generation very easy [Ora].
The Apache Synapse configuration XML schema can be downloaded from the project page[2] and it is also packaged in the source files. The Apache Synapse schema is very complex and there are several issues while trying to compile the Java model:

- **Structure mapping**
  There are several problems with mapping *xs:all* elements in the Synapse configuration combined with *xs:any* elements. These can be manually solved by relaxing the quantity restrictions in these constructs or using *xjc* binds that solve the issue. These changes are leading to equivalent parsing results.

- **Java inner class naming problem**
  The *endpoint.xsd* contains valid structures that cannot be represented in Java code. The failover and failure endpoint were modeled as group of endpoints with additional attributes. That generated an endpoint class with two inner Java classes that all share the same name. This can be valid but causes problems if the fully qualified class name has not been used. To avoid these problems there is a workaround in the source code that the two mentioned endpoints are exported in own XSD files. Then the *xjc* will compile the schema using correct naming.

These changes result in a useful parsing binding from JAXB for the Synapse configuration. But the generated objects can have issues with marshalling the Java objects to an XML representation. This points to people who possibly want to modify the crawler application of this thesis, to use it as a backend for an Apache Synapse configuration creation tool. It is possible, but the documented modifications in the schema have to be validated for marshalling.

[2]http://synapse.apache.org/ns/2010/04/configuration/synapse_config.xsd

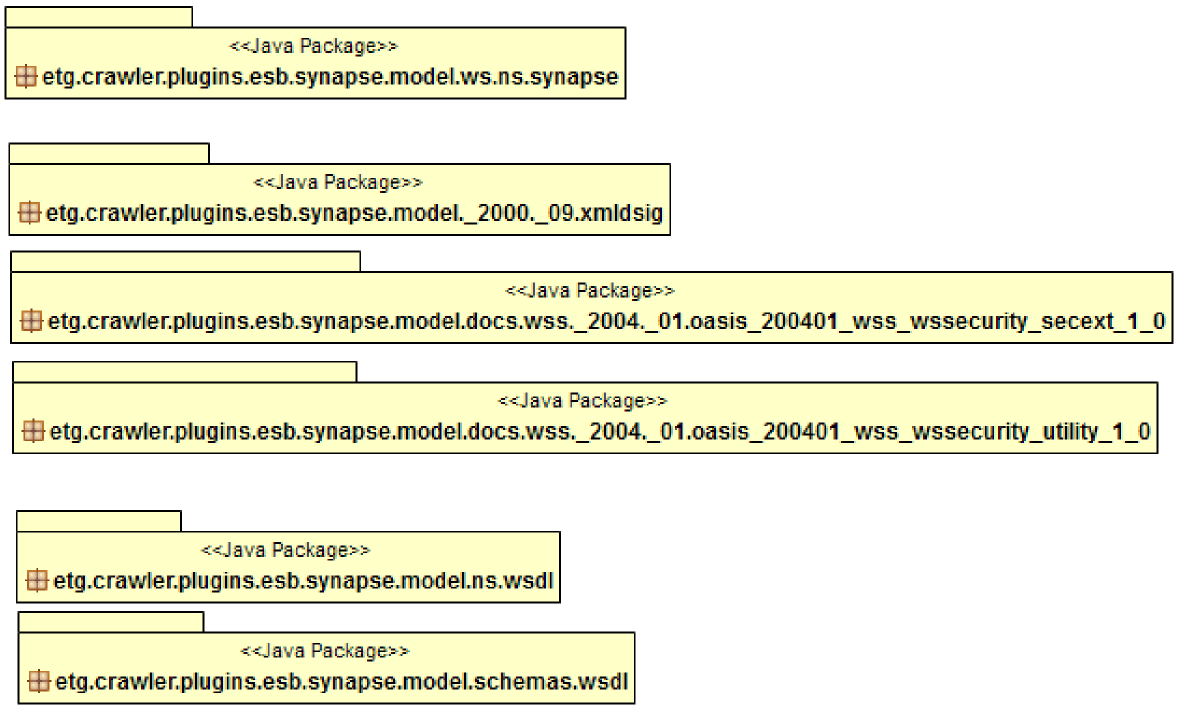

<<Java Package>>
etg.crawler.plugins.esb.synapse.model.ws.ns.synapse
<<Java Package>>
etg.crawler.plugins.esb.synapse.model._2000._09.xmldsig
<<Java Package>>
etg.crawler.plugins.esb.synapse.model.docs.wss._2004._01.oasis_200401_wss_wssecurity_secext_1_0
<<Java Package>>
etg.crawler.plugins.esb.synapse.model.docs.wss._2004._01.oasis_200401_wss_wssecurity_utility_1_0
<<Java Package>>
etg.crawler.plugins.esb.synapse.model.ns.wsdl
<<Java Package>>
etg.crawler.plugins.esb.synapse.model.schemas.wsdl

Figure 4.3: Illustration of the create packages from JAXB xjc

Figure 4.3 shows the six created packages by *xjc*. JAXB uses naming conventions that takes the XML namespaces of the schema files as package names. Only the first package *etg.crawler.plugins.esb.synapse.model.ws.ns.synapse* contains Apache Synapse objects. The other packages are imported schema that are used inside the Synapse data types. For example the *etg.crawler.plugins.esb.synapse.model.ns.wsdl* package consists of the WSDL definitions like *portType*, *binding* and *service*. Finally the *Definition.class* is used as XML root element for the initialization of the *JAXBContext*. The context creates the unmarshaller that can parse and create the Java objects containing the routing information.
The *Definition.class* also contains the routing information in the *Proxy.class* and *NamedSequence.class*. The children of these classes are the mediators (cf. section 4.3.5) and endpoints. Figure 4.4 shows the high level routing information objects.

4.8.2 Crawling Static Routing Information

The crawling algorithm is using the generic data model discussed in Chapter 5. Briefly the model contains of several generic types with defined semantics. There are endpoints, gateways, gateways with conditions and mediators. These Entities are designed to reflect any ESB structure.
The crawler first starts to get all sequences (cf. Section 4.3.1) and proxys (cf. Section 4.3.2) of the Apache Synapse configuration. Both of them we will now refer to as routes. Basically the crawler first iterates through all routes.

Listing 4.7 Starting the RouteBuilder with proxy and routes

```
for each Sequence ∈ Definition do
      abstractRoute ← routeBuilder(sequence.getMediatorList())
}
for each proxy ∈ Definition do
      abstractRoute ← routeBuilder(proxy.getMediatorList())
}
```

Listing 4.7 shows that for each element the *routeBuilder(List<Mediator> mediators)* function is called. The JAXB binding provides the *getMeditatorList()* that returns all children mediators. The *RouteBuilder* is iterating on the first level elements of the Apache Synapse configuration file.

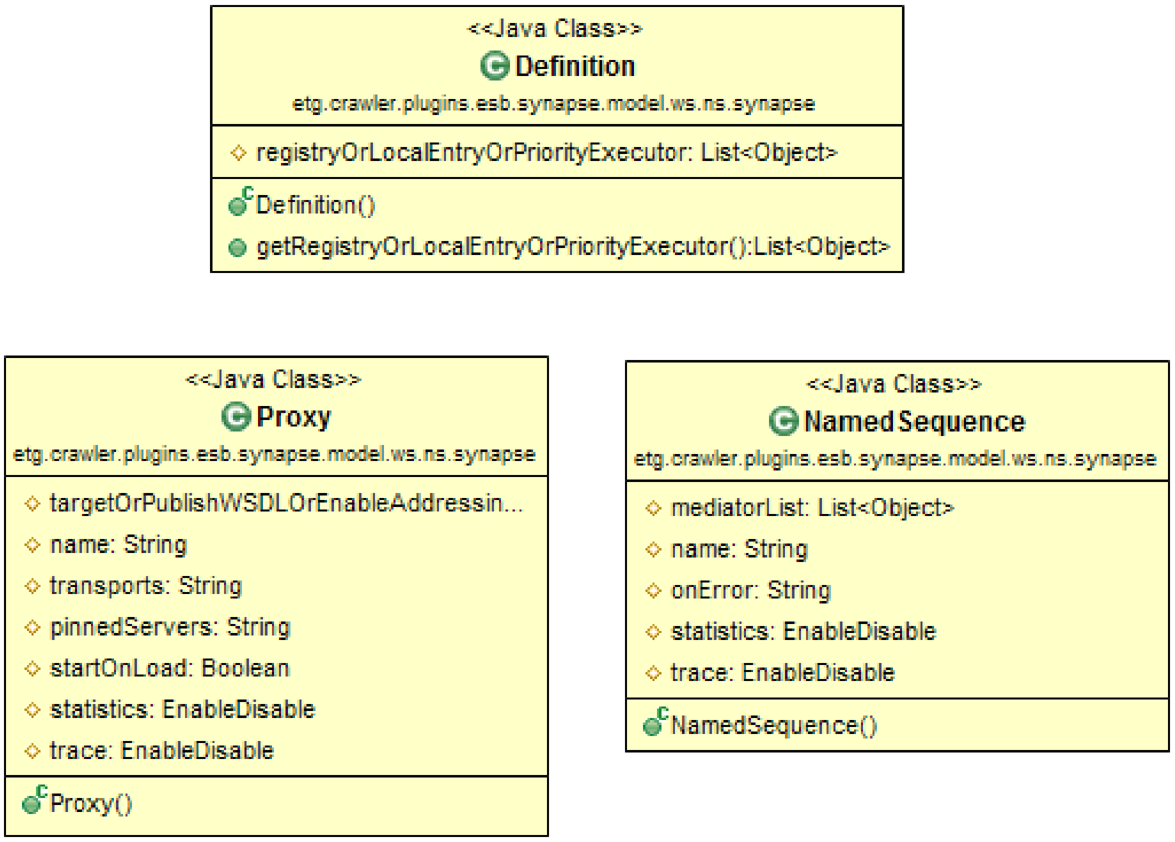

<<Java Class>>
Definition
etg.crawler.plugins.esb.synapse.model.ws.ns.synapse
registryOrLocalEntryOrPriorityExecutor: List<Object>
Definition()
getRegistryOrLocalEntryOrPriorityExecutor():List<Object>
<<Java Class>>
Proxy
etg.crawler.plugins.esb.synapse.model.ws.ns.synapse
targetOrPublishWSDLOrEnableAddressin...
name: String
transports: String
pinnedServers: String
startOnLoad: Boolean
statistics: EnableDisable
trace: EnableDisable
Proxy()
<<Java Class>>
NamedSequence
etg.crawler.plugins.esb.synapse.model.ws.ns.synapse
mediatorList: List<Object>
name: String
onError: String
statistics: EnableDisable
trace: EnableDisable
NamedSequence()

Figure 4.4: The definition of the message flow elements proxy and sequence

Listing 4.8 Route Builder algorithm building the AbstractRoute

```
function appendChild(parentElement)
for each element ∈ parentElement.getMediatorList() do {
      if(isMediator(element) {
            element ← assembleMediatorInformation(element)
            parent.addChild(element)
      }
      if(isGateway(element) {
            element ← assembleGatewayInformation(element)
            parent.addChild(element)
      }
      if(isEndpoint(element) {
            element ← assembleEndpointInformation(element)
            parentElement.addChild(element)
      }
}
```

The pseudo code in Listing 4.8 shows that the input can be of the type: (i) mediator, (ii) gateway or (iii) endpoint. Condition elements are handled by the gateways.
Depending on the generic model semantics the functional components have been categorized. They have a strong correlation to the gateway element. The returned element will be added as child of the parent. This function is crawling the graph recursively while checking each element of the Apache Synapse configuration for additional element children. For each mediator pattern there is an implementation in the *assembleMediatorInformation()* function. There are two examples illustrated in Listing 4.9.

Listing 4.9 Information Assembler

```
function assembleMediatorInformation(Object obj)
      if (obj instanceof Log) {
            Log element = (Log) obj;
            addMediatorProperty("Level", element.getLevel().toString(), element);
            return abstractMediator;
      }
      if (obj instanceof Send) {
            Send element = (Send) obj;
            if (element.getEndpoint() != null{
                  abstractMediator.setChild(assembleEndpInfo(element.getEndpoint()));
            }
            else {
                  addMediatorProperty("to:Endpoint:", "message 'To' header",
                      iator);
            }
            return abstractMediator;
      }
}
```

Generally every element is assigned to the role mediator, gateway or endpoint. The log element is a simple mediator that cannot have any children. So the crawler returns the element without calling another assemble function or calls the *routeBuilder* again. On the other hand the *send* element has two possible structures:

- There exists no children only a <*/send*> element
  So there are no children to add at this element. After adding the properties the send mediator will be returned.
- If there are endpoint child elements
  The send element adds an endpoint child that is being assembled by *assembleEndpointInfo(element.getEndpoint())* function.

This individual structure and syntax can be compared to Section 4.3. The syntax outline defines the possible children of the Apache Synapse configuration element. The possibility of a *mediator+* or *endpoint?* structure is reflected by an assemble function in the code. The recursive crawler will stop if there are no child structures in the outline. This implementation is easy to maintain. Because if Synapse change a configuration element then only the specific logic has to be adapted. Also the logic atomically bound to the specific assembler step. If the log mediator would enabled child elements in the future, then only the specific assemble logic would change while the rest of the code will be untouched.

4.8.3 Crawling Routing Statistics

The dynamic routing statics cannot be found in the configuration files. It would only be possible to estimate the routing in a limited way. For example a *round robin load balancer* would obviously invoke the endpoints proportionally. But the number of client calls would be unknown.
Because of this limitation it is necessary to gather real routing information from the Synapse instance. The Synapse JMX (cf. Section 4.7) enables the crawling of endpoint statistics. This covers the received messages and faults. The statistics are bound to the endpoint name that has been defined in the Synapse configuration file.
Synapse enables JMX by default and it can be connected with an arbitrary JMX client. The implementation uses the *MBeanServerConnection* provided by Java. Then the *MBean* of Synapse can be queried with the object name *org.apache.synapse:Type=Endpoint,**. This returns all JMX interfaces of all used endpoints of the Synapse instance. Finally the attributes *MessagesReceived* and *FaultsReceiving* contain the statistics. They will be added to the *endpointStatistics* object. It contains the endpoint statistics correlated to the name. The *assembleEndpInfo()* function adds these information to the properties of the endpoint object.

4.9 Discussion of Crawling Results

The crawler tries to create an exact snapshot of the structure of Synapse. Every functional component can be crawled and mapped to the generic data model. But there are also some limitations and adaptions to enable the transformation:

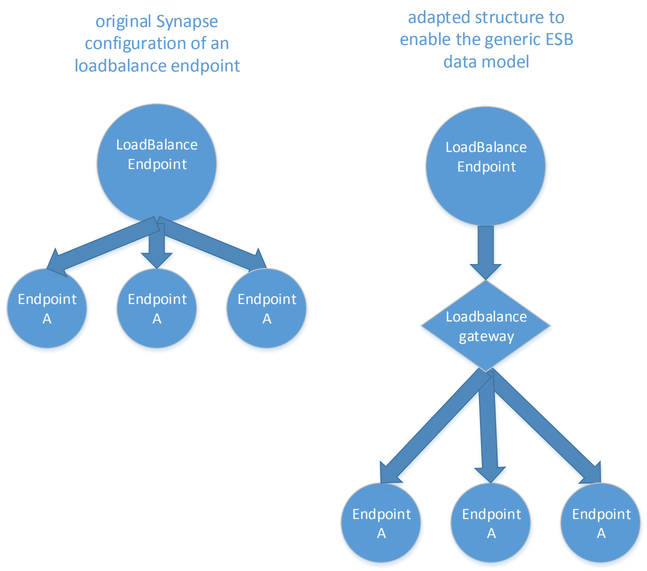


Figure 4.5: Transformation example of the Apache Synapse model to the generic ESB data model

- **Endpoint semantics**
  The generic data model defines that an endpoint can only have one child. Multiple endpoints are only possible with the gateway element. Figure 4.5 illustrates the transformation to the generic data model. The loadbalancer is still an endpoint but is connected to a gateway with the type loadbalancer. The gateway has no conditions so the semantics are clear that the gateway does not contain any routing logic. The gateway dispatches the message to the given endpoint.
- **Remote registries**
  Apache Synapse allows to load configuration from a remote registry. These data location is not accessible for the crawler. They will be ignored.
- **Dynamic routes**
  Synapse can configure elements such as dynamic sequences and endpoints. They can be changed with external scripts or XSDs in the registry. The Synapse crawler cannot cover these dynamical changes and only takes a snapshot of the current routing structure while accessing the Synapse configuration files.
- **Custom mediators**
  The prototype of the Apache Synapse crawler cannot detect custom mediators. They can be implemented using Java. It is needed to plugin the Java class into Apache Synapse. The crawling would see the last predecessor of the custom mediator as a dead end. Fortunately custom mediators can also be implemented using the extension mediators

(cf. Section 4.3.8). These mediators can be handled by the Synapse crawler and correctly processed.

- **Concatenation of routes**
  As already discussed in the interpretation of the Camel crawling results 3.9 the correlation of routes will not correlated in the resulting generic data model structure. There are the same issues like multiple interpretation alternatives. The crawler only changes the structure if the transformation is inevitable. The crawler can also be considered lazy because it will only rebuild the structure found in the configuration and never interlink them.

CHAPTER 5

GENERIC MODEL FOR ENTERPRISE SERVICE BUS ROUTING

The research gave an structural insight of the ESBs Apache Camel and Apache Synapse. They both implement a routing with enterprise integration patterns compatible structures. For developers of plugins for the ETG Framework it would be handy to adapt a model that automatically can be imported to the Enterprise Topology Graph. The Synapse crawler discussed in the Chapter 4 directly adapts the generic model. The crawler implementation adapt the concept of the model entities and transform the inner structure accordingly.

5.1 Usage and Requirements

The Generic Data Model for Enterprise Service Bus (GDMESB) should assist ETG Framework developer implementing crawler plugins to complete enterprise topologies with routing information. The GDMESB can be used:

- to assist developers to easily integrate their crawler API to the ETG Framework.
- to map ESB platform entities with generic semantics that can be compared between different ESB products
- to provide an automatically transformation from the GDMESB to the ETG model.

The structure of the crawled data can be very different. The routing information are generally represented by a concatenation of routing entities. Despite of the variably of the data the GDMESB have to enable a conceptual schema in a generic way. That requires several GDMESB characteristically requirements to the model:

R1 The model structure describes routing logic as a directed graph.

R2 The model structure allows free combination of entities if the entity itself provides the possibility to do so.

R3 The model entities have defined orchestration purposes. Those cannot be combined in ways that disagree with their semantics.

R4 Each model entity has unique semantics that distinguishes itself from every other one.

R5 Every model entity allows custom properties and notations. This includes the denotation of the entity type.

These requirements ensure the compatibility with ESB routing information. It also ensures a precise transformation to the ETG model.

5.2 Model Specification

5.2.1 Model Structure

The GDMESB is designed for describing ESB routes. Theses routes typically represent an mediator flow in the Enterprise Service Bus. Typically they are structured as a directed graph. Generally the GDMESB will adapt this model and also represents a message flow as a chain of mediators. This enables flexible structure containing abstract entities. These model elements can also be called abstract model elements. The graph structure is created with a child/parent between those elements. Also the combination of the abstract model elements is limited to provide correct usage of the model entities. The user can create a generic representation of a message flow with route, mediator, gateway with condition, endpoint and proxy elements.

5.2.2 Entity Descriptions

Every entity of the GDMESB has a defined purpose and functionality.

- **Route**
  An ESB can have several routes defined. A route defines chain of routing elements that perform some processing steps. The mediator of the ESB uses the logic defined in the route to create a message flow.
  This is considered as the root element of all following GDMESB entities. But it can also be used as a reference to another reusable route. A route has only one child that describes the entry point to the route.
- **Mediator**
  There are several processing steps designed typically as pipe and filters on a route. The mediator represents a filter and the child connection pipes usually to another mediator or an endpoint. This entity represents processing semantics on the message of the route.
- **Gateway**
  A gateway is a dispatcher of messages in the Enterprise Service Bus. It is the only entity that can have multiple children. It does not do any processing as it only doing the

routing. The gateway can implement the routing logic or the gateway can be upstream connected by a mediator or endpoint.

- **Routing Condition**
  A routing condition can be a child of the a gateway or mediator. It annotates a route with a conditional expression. As an example a filter mediator could check messages based on the content. If the content matches the condition it will sent to the next mediator. Otherwise the message will be dropped. Then the filter mediator would have a gateway with a if condition and an else condition.

- **Endpoint**
  The backend service invoked by the ESB system is called an endpoint. It is doing request processing from the outside client application. The service can be any executable program or Web service.

- **Proxy**
  The proxy is the exposed service of a route. Like Synapse the routes can only be invoked by a proxy endpoint that routes the message to the internal backend endpoint. There is only semantically difference here because ETG Framework users could want to differentiate between those endpoints.

5.2.3 Entity Characteristics

The entities have restrictions and properties that have to be satisfied. Thus the restrictions can be used to validate the correctness of a generic data model. The characteristics are flexible as possible to ensure compatibility with several ESB model structures. For every entity type Table 5.1 lists following characteristics:

Entity type	Number of child-relations	ETG Type	occurence (max)	Possible predecessors
Route	(1,1)	ESBmodel_Route	1	unrestricted
Mediator	(0,1)	ESBmodel_Mediator	N	unrestricted
Gateway	(1,n)	ESBmodel_Gateway	N	endpoint or mediator
Routing Condition	(1,1)	ESBmodel_RouteCondition	N	gateway
Endpoint	(0,1)	ESBmodel_Endpoint	N	unrestricted
Proxy	(0,1)	ESBmodel_Proxy	1	route

Table 5.1: Characteristics of the generic model entities

- **Number of child relations**
  The GDMESB entity relations are modeled with parent child relations. The number of child relations depends on the semantic of the entity. For instance, only the gateway can have multiple children.
- **ETG type**
  This property is needed to address the entity in the ETG Framework.
- **Occurence**
  There is a restriction how often an entity can be used in a route structure. This includes the route and proxy that are only allowed once in the route.
- **Possible predecessors**
  The semantics of the entities sometimes restrict their usage in the route relation. The gateway is built for routing. The routing logic has to be modeled with an endpoint or mediator. Hence the gateway can only be a child of an endpoint or mediator. The route condition is a tool to model control structures. Only the gateway can have child routing conditions. Finally the proxy can only be the second optional element in the structure.

The ETG Framework plugins have to ensure theses characteristics to use the transformation to the builder. It also ensures the correct usage of the entities.

5.2.4 Best Practices

Enterprise Service Buses can utilize the enterprise integration patterns that support complex integration of applications. It is a consistent vocabulary and a visual notation to describe large-scale integration solutions. They represent the core language of EAI and enable a definition of ESB flows.
Many of the route elements of Apache Synapse and Apache Camel can also be mapped to the pattern elements. The Table 5.2 shows the mapping between the EAI patterns and how they can be assembled with the GDMESB. These can be considered as usage recommendation. Furthermore the ETG Framework plugin developer will probably have to do various adaptions. This is anticipated because the model is designed to be flexible. The prototype of the Apache Synapse adapts these best practices.

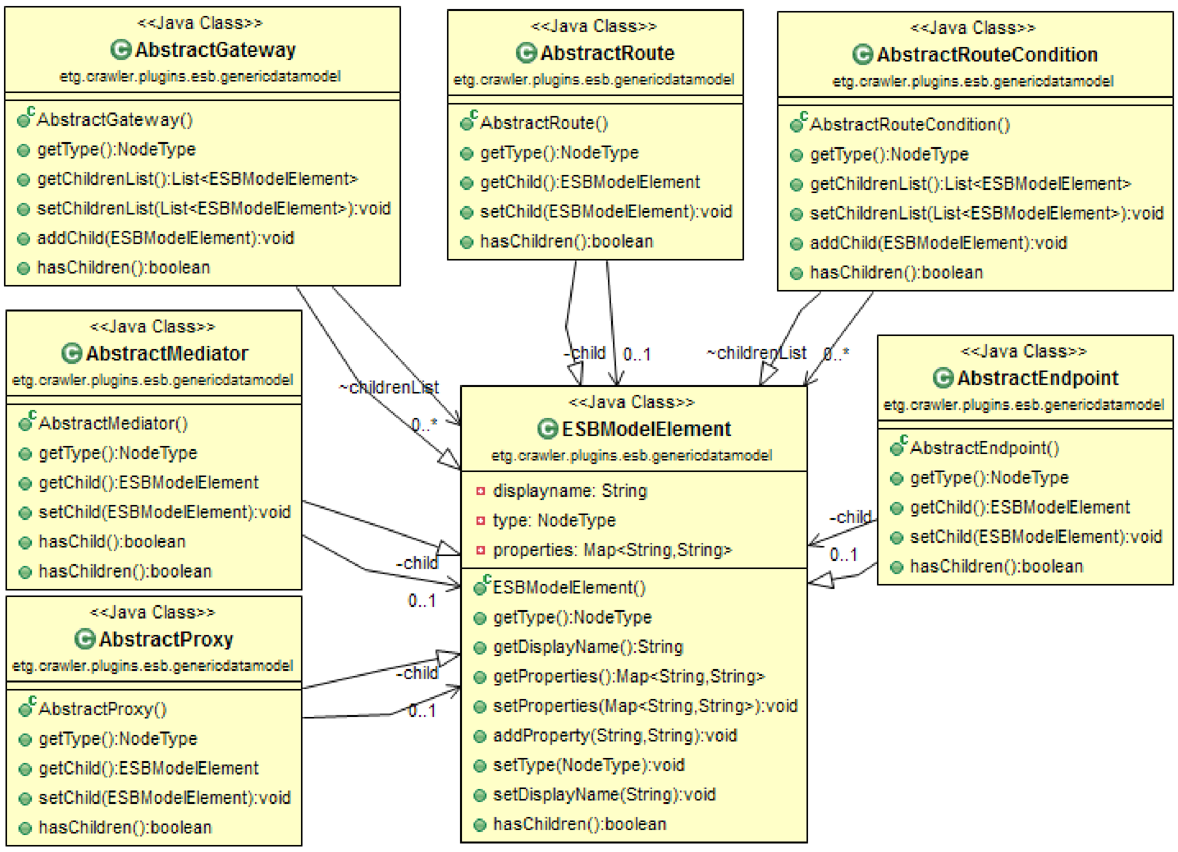
Messaging Systems	
Message Router	one mediator with gateway as child
Message Translator	one mediator
Message Endpoint	one endpoint
Message Routing	
Content-Based Router	one mediator with gateway as child
Message Filter	one mediator
Dynamic Router	one mediator with gateway as child that has several routing conditions
Recipient List	one mediator with gateway as child
Splitter	one mediator
Aggregator	one mediator
Resequencer	one mediator
Composed Message	splitter mediator followed by router mediator that connects a gateway. The gateway has several elements as children for the submessages. These elements all have the aggregator as child.
Scatter-Gather	content based router mediator with a gateway. Every child element of the gateway is referring to the aggregator.
System Management	
Control Bus	separate route with the control bus logic
Detour	Gateway with two children that reference the default route and the detour route
Wire Tap	route element with the reference to another route
Message History	one mediator
Message Store	one mediator
Smart Proxy	mediator with id property for the request and response message path

Table 5.2: Mapping between Enterprise Integration Patterns and Generic Data Model elements

5.3 Interpretation and Design

5.3.1 Data Model

The Generic Data Model is implemented and accessable in the ETG Framework project. It provides data classes that represent the entities discussed in this Chapter. The Frameworks also implement a transformation to convert a GDMESB to an ETG. This enables easy integration of Enterprise Service Bus structure.
Every model entity extends the abstract *ESBModelElement*. As discussed every entity has a type, an element and properties defined. They are already provided by this class. The functions are derived from the *ESBModelElement* interface. It covers all the *getter* and *setter* functions to common function signatures across every entity.



<<Java Class>>
AbstractGateway
etg.crawler.plugins.esb.genericdatamodel
AbstractGateway()
getType():NodeType
getChildrenList():List<ESBModelElement>
setChildrenList(List<ESBModelElement>):void
addChild(ESBModelElement):void
hasChildren():boolean
<<Java Class>>
AbstractRoute
etg.crawler.plugins.esb.genericdatamodel
AbstractRoute()
getType():NodeType
getChild():ESBModelElement
setChild(ESBModelElement):void
hasChildren():boolean
<<Java Class>>
AbstractRouteCondition
etg.crawler.plugins.esb.genericdatamodel
AbstractRouteCondition()
getType():NodeType
getChildrenList():List<ESBModelElement>
setChildrenList(List<ESBModelElement>):void
addChild(ESBModelElement):void
hasChildren():boolean
<<Java Class>>
AbstractMediator
etg.crawler.plugins.esb.genericdatamodel
AbstractMediator()
getType():NodeType
getChild():ESBModelElement
setChild(ESBModelElement):void
hasChild():boolean
hasChildren():boolean
<<Java Class>>
AbstractProxy
etg.crawler.plugins.esb.genericdatamodel
AbstractProxy()
getType():NodeType
getChild():ESBModelElement
setChild(ESBModelElement):void
hasChildren():boolean
<<Java Class>>
ESBModelElement
etg.crawler.plugins.esb.genericdatamodel
displayname: String
type: NodeType
properties: Map<String,String>
ESBModelElement()
getType():NodeType
getDisplayName():String
getProperties():Map<String,String>
setProperties(Map<String,String>):void
addProperty(String,String):void
setType(NodeType):void
setDisplayName(String):void
hasChildren():boolean
<<Java Class>>
AbstractEndpoint
etg.crawler.plugins.esb.genericdatamodel
AbstractEndpoint()
getType():NodeType
getChild():ESBModelElement
setChild(ESBModelElement):void
hasChildren():boolean
~childrenList
0..*
-child
0..1
~childrenList
0..*
-child
0..1
-child
0..1
-child
0..1

Figure 5.1: Base call of all generic data model entities

The class diagram in Figure 5.1 displays all objects of the generic data model. Every parent object can only add a *ESBModelElement* to ensure type safety.

5.4 ETG Builder

The ETGBuilder exports the structures of the GDMESB in the ETG graph. It performs a worklist algorithm that iterates over a worklist of discovered route elements. Every processing step of one element will be checked for children and then appended to the ETG model. The route builder needs four input variables:

- **WrappedETG**
  This entity represents the current state of the ETG graph.
- **ICrawlerPlugin**
  The Plugin implementation that uses the *ETGBuilder*. The ETG Framework force to indicate the responsible plugin for change operation on the graph.
- **WrappedNode**
  The parent node that will be connected to the first route element.
- **AbstractRoute**
  The GDMESB route object that is the first input for the algorithmn. Every entity will depend on this with a *is subset of* relation (cf. Section 6).

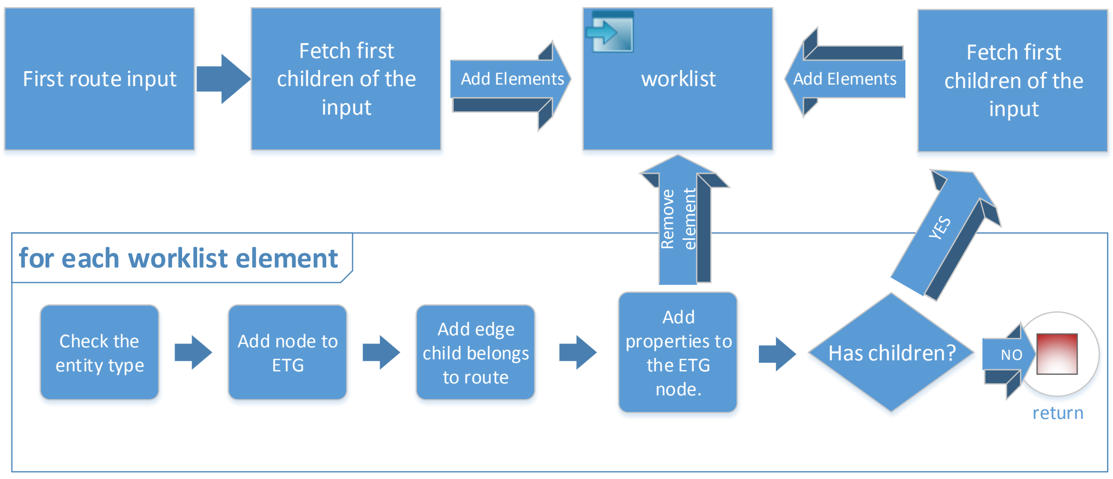

First route input
Fetch first children of the input
Add Elements
worklist
Add Elements
Fetch first children of the input
Remove element
YES
for each worklist element
Check the entity type
Add node to ETG
Add edge child belongs to route
Add properties to the ETG node.
Has children?
NO
return

Figure 5.2: Illustration of the processing steps of the ETG route builder.

After creation of the *ETGBuilder* object with the input data, the import can be started with the build() method that returns the modified ETG graph. The simplified process of the method is illustrated in Figure 5.2. The work contains the following processing steps:

1. Process the route element and add it to the ETG.
2. Initialize the worklist with the children of the route element.

3. Get one worklist item and start the processing loop:

   Processing and transformation of an discovered child element

   a) Check the entity type of the input. There is a custom implementation for each entity of the add methods.
   b) Add the route to the ETG
   c) Add the edges to the parent route (*depends-on* relation) and the father child relation to the ETG.
   d) Export the properties in the create ETG node.
   e) Remove element from the worklist.
   f) Has the element children? If yes add them to the worklist. If not return.

4. Loop until the worklist is empty.

Finally the model has been transformed to the ETG model. The structure of the resulting ETG model is part of the Chapter 6. The iterative worklist algorithm has some limitations. There could be memory issues if the worklist was very large. This would need millions of entities in one route. However one message flow does typically not contain a large set of entities, in fact they rarely exceed hundred entities.

CHAPTER 6

ETG FRAMEWORK INTEGRATION

6.1 Enterprise Service Bus Discovery

Before invoking the Enterprise Service Bus crawler it is necessary to detect a possible ETG instance in the enterprise IT topology. Typically ESB are exposing their functionality with addressable endpoints. For instance in Apache Synapse they are called proxy endpoints. Clients typically invoke them with HTTP request.

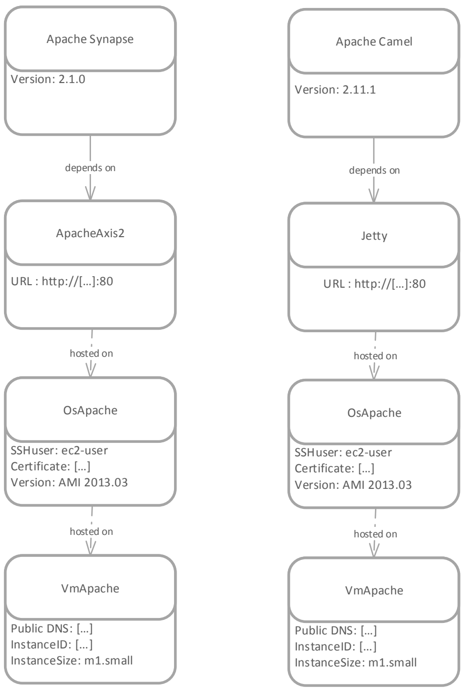


Figure 6.1: Synapse and Camel application topology rendered without routes as Enterprise Topology Graph. (following the *Visual Notation for* TOSCA) [BBLS12]

Like the Web Server Plugin the HTTP request usually contains a server header that can contain more details of the application. This concept has already been tested and implemented by Jakob Krein [Kre12]. The ESB can be added in the same iteration like the applications. Because it typically depends on a server that is running the application. Basically an ESB product will be discovered in the same step like PHP modules or web applications. Figure 6.1 shows an example ETG and illustrates Apache Synapse and Apache Camel in a sample enterprise IT structure.
Typically an ESB depends on a transportation application that invokes the routes. If the ESB itself represents the Web server than they would be hosted on an operating System like Windows or Linux. This is not the case with Apache Camel and Apache Synapse. They directly depend on a Web server engine that appears as standalone from the outside.

Listing 6.1 HTTP Response header of an HTTP Request

```
[Status-Line]           HTTP/1.1 200 OK
Content-Type:           text/xml; charset=UTF-8
Date                          Tue, 24 Sep 2013 12:55:21 GMT
Server                        Synapse-HttpComponents-NIO
Transfer-Endcoding      chunked
Connection                    Keep-Alive
```

The Listing 6.1 illustrates a response header of Apache Synapse. The Service is hosted on an Axis2 server that directly invokes the Synapse instance. The Server field indicates a running ESB instance. Based on the discussed fact that most ESB depend on their web server component, they usually run in the same instance and can be addressed with the same URI as proxy service.
The *NetworkHelper* of the ETG Framework can be used to analyze the content of the HTTP messages. If a HTTP field hints on a running ESB the ESB plugin crawler will investigate the URL.

6.2 ESB Structure Rendering in the ETG

Enabling Enterprise Topology Graphs to represent routing information the Framework has to be enhanced with nodes and edges that suit the semantic of an ESB flow. The ETG types are based on the generic data model for ESB routes. The nodes can be categorized as ESB product and route element types. For supported ESBs like Apache Synapse and Apache Camel there is a derived node type like *ESB_Camel* and *ESB_Synapse*.
The catalog of available route elements to model an ESB structure is the same as the entities of the generic data model. The properties and requirements of Section 5.1 will also be applied in the ETG model. The Apache Synapse plugin shows the successful usage of the model in the core of the crawler API. On the other hand Apache Camel uses a custom *CamelStructure* model that was also successful converted in the ETG model.
The route node types *ESBmodel_Route*, *ESBmodel_Endpoint*, *ESBmodel_Proxy*, *ESBmodel_Mediator*,

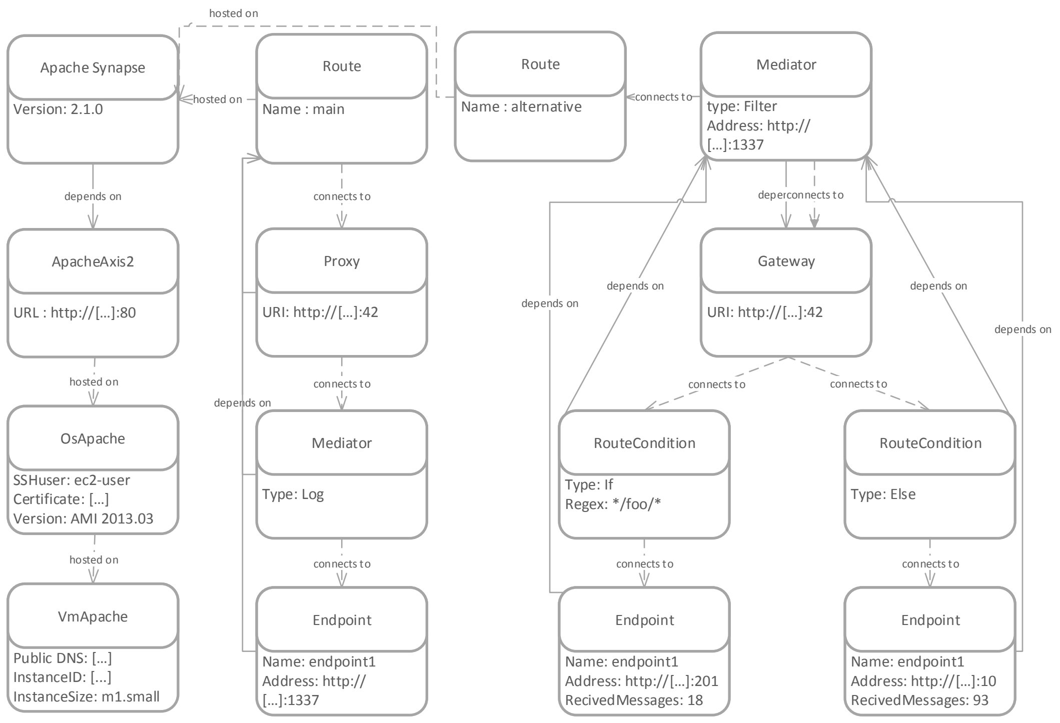


Figure 6.2: Example of an complete Apache Synapse topology with two routes illustrated as ETG

ESBmodel_Gateway and *ESBmodel_RoutingCondtion* can be used to complete ETGs with routing information. Figure 6.2 illustrates a complete example of two Apache Synapse routes.
The usage of the edges: *depends-on*, *connected-to* are also illustrated in the picture. Every route element node is connected to the route node. This enables to determine which route element belongs to which route. Eventually this could be interesting for query purposes.
However to ensure correct edge connection of the nodes the plugin developer should use the *ETGBuilder*. Otherwise the following relation rules should be applied:

- Every route element node needs to have a parent route node. The reflexive depended on relationship on the route node itself should not be applied.
- Only the route node depends on the ESB node.
- Every node can be connect to one or zero nodes. No reflexive relationships are allowed.
- Only the gateway node has multiple children with a *connect-to* relation.

6.3 Apache Camel Plugin

The Plugin for Apache Camel is based on the JMX driven crawler plugin. The only needed information to connect to the management interface is the IP address. But the discovery is more complicated because Camel is not correlated with the response HTTP headers. The Apache Synapse plugin can determine a Synapse instance based on these information. The only supported Web server that can handle HTTP requests is provided by the Camel *jetty-component* (cf. Section 3.5.3). Every discovered Jetty instance of the ETG Framework has to be tested if there is an Apache Camel ESB hosted on the server. There is no alternative to this try and error solution. Because Camel can run in an application server or as standalone Java program on any operating system. It would be possible to check the running processes in the operation system, for instance with the *top* command in Linux. The more viable way is to try a Camel connection to every found Jetty URL.
There is a connection tester implemented in the *CamelConnecter* utility in the API. If the utility fails to connect the plugin assumes that the Jetty instance is deployed without an Apache Camel.
The Camel API returns a *CamelStructure* object which is not consistent with the in Section 5.1 mentioned requirements of the generic data model for the ETG graph. So there is a transformation needed. After that process the *ETGBuilder* applies the camel routing information to the ETG.

6.4 Apache Synapse Plugin

The Apache Synapse plugin uses the Synapse Crawler API discussed in section 4.8. The crawler needs access to the configuration files of Apache Synapse. Their location is defined and can be accessed in the path *[SYNAPSE_HOME]/repository/conf*.
The Server must provide an SSH access to the file system of the Synapse instance. The ETG Framework enables to run scripts on the operation system. The *SSHHelper* provides an execution for bash scripts. For the discovery of the Synapse configuration files the *"fgrep -rl –include=*.xml "definitions xmlns=ḧttp://ws.apache.org/ns/synapse" *"* command. This is a fast and very flexible way to get the configuration files with matching configuration definitions.
The command line utility grep is mandatory on Linux systems. The program searches files that match a regular expression defined by the user. The plugin prototype assumes that a Windows server also uses grep.
Finally paths containing sample files will be filtered out of the result string. The string now contains the locations of the Apache Synapse configuration files. If the files are accessible they can be passed to the JAXB XML unmarshaller(cf. Section 4.8.1). It invokes the *RouteBuilder* for every route and proxy found in the XML.
The Synapse plugin uses the Synapse crawler API to get the statistical result. As mentioned in 4.8.2 the API uses the generic data model for describing ESB routes (cf. Chapter 5). So the route information can be imported using the *RouteBuilder* transformation functionality.
The dynamic routing information will be gathered with the *SynapseStatistics* tool in the API package. It uses the JMX Interface of Synapse to gather the data. The management support is

reachable by default with the connection string *service:jmx:rmi:///jndi/rmi://:/synapse* for Apache Synapse. The detailed implementation has been discussed in the Section 4.8.3.

6.5 Evaluation

Both plugins were evaluated in several steps. Firstly the crawling API was tested, secondly the model transformation was evaluated and then finally the plugin itself.
Firstly the examples of Apache Camel were analyzed. Because of the variety of implementation of the examples they would be hard to deploy on a test system without needing a lot of customization. To avoid this the complete business coffee shop example (cf. Section 3.6.2) has been chosen. First the crawling entities of the JMX output have been analyzed and then compared with the Apache Camel structure. The prototype can crawl all patterns mentioned in Section 3.3. Even unknown patterns will be correctly appended to the ETG with the JMX output as property. So the Camel Crawler is capable to fetch any Apache Camel structure. Both examples produced correct results in the ETG Framework.
Secondly the Apache Synapse Crawler API was evaluated against every mediator and proxy example contained in the Synapse source. Overall, there are 18 examples that have been successfully tested. Based on the fact that the configuration of Apache Synapse is formalized in machine readable XML they could be tested without deploying a Synapse instance. The information gathering method of parsing configuration files also ensure correctness among different deployment scenarios. The examples mentioned in Section 4.6 were tested in a running Synapse instance using the ETG Framework.
The evaluation of the *ETGBuilder* ensures correctness of the transformation and representation of the routing information in the ETG. The builder was tested with random generated test input and indirectly tested with the examples of both ESBs covered in this thesis.

CHAPTER 7

SUMMARY AND OUTLOOK

In recent years, new technologies have been developed to face the challenges of todays businesses. Enterprise IT is trying to improve and re-engineer their internal IT structure using state of the art hardware and software solutions. Enterprise Application Integration supports the integration of application build on different kind of platforms. The Enterprise Service Bus is a central part of todays modern SOA. Before this transformation can begin, insight to the structure and components is crucial. Even after reorganization of the enterprise IT further iterations of optimization are needed to reduce costs, enhance flexibility and improve the quality of service of the whole business. Hence, continuously documentation of the complete enterprise IT including state of the art technologies are important for the analysis and optimization. The ETG Framework provides automated discovery and maintenance of Enterprise Topology Graphs. Each Framework plugin is specialized on different components and all their dependencies. This thesis enhances the discovery framework with the fundamentals, concepts and implementation for the extraction of ESB routing information to complete enterprise topologies.
First, the fundamentals have been researched based on the ESB products Apache Camel and Apache Synapse. Both ESBs conform with the ESB characteristics defined by D. Chappell, while following different conceptual approaches and architectures. Apache Synapse directly implements the Enterprise Integration Patterns using a Java and an XML description language. Apache Camel is basically only a mediation engine that has to be extended with Web service technologies to provide full ESB functionally. Because of the simplistic architecture Apache Camel is the routing engine of several ESB products like Apache ServiceMix and Fuse ESB Enterprise. Therefore, the research of this thesis can also be adopted on these products. After all Apache Synapse provides a huge catalog of fine-grained routing entities enabling enterprise integration. It is a complete lightweight Enterprise Service Bus also available as commercial derivative called WSO2 ESB. The thesis presents different concepts tailored for both of the products. They are implemented as reusable application utilities and represent the prove of concept for these approaches. All in all, two information gathering concepts have been tested and successfully applied. The Apache Camel plugin is using the sophisticated JMX management interface which supplies raw routing data requiring large processing efforts. On the other hand, the Apache Synapse plugin is using a mix of parsing XML configuration files with JMX as statistical routing information source leading to complex parsing logic. All

in all, Java JMX technology provides a distributed and modular solution providing real time static and statistical information that fits perfectly for our purpose.
During the implementation the possible benefits of an generic data model for Enterprise Service Buses became obvious. The elaborated data model in this thesis can be used to describe ESB routing information using flexible message flow elements. Future ESB plugin developers will benefit from the provided automated transformation to the ETG model.
The evaluation has been done with suitable test scenarios that reflect workflows and ESB configurations with Web service endpoints. The examples have been choose and modified to cover the core functionality and provide effective testing of the concepts and prototypes of this thesis.
Future work could evaluate the transformation to the ESB generic data model based on different ESB vendor models. This is needed to evaluate the flexibility of the data model for any ESB structure. Also ESB developers could provide a native ESBGDM export in their application. The created plugin prototypes and concepts should also be tested in real world scenarios. The concepts of this thesis could also be adapted to integrate more ESB products as ETG Framework plugin.

APPENDIX A

SYNAPSE DEPLOYMENT AND INSTALLATION GUIDE

This chapter explains how to run a Synapse ESB. It is a compact summary of the official quick start guide[1] on the Apache Synapse Project page. The guide assumes the use of a Windows system.

A.1 Deploying Apache Synapse

A.1.1 Pre-requisites

The following software should be installed on the host system:

- Java JDK
  Download: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Apache Ant
  Download: `http://ant.apache.org`
- Apache Synapse Release Binary Distribution 2.1.0
  Download: `http://synapse.apache.org/download/2.1/download.cgi`

After that extract Synapse to a directory of your choice that we now will refer as SYNAPSE_HOME

[1] `http://synapse.apache.org/userguide/quick_start.html`

A.2 Step by Step Guidance

1. Running Axis2 Server

   ```
   SYNAPSE_HOME/samples/axis2Server/axis2server.bat
   ```

   You can check `http://localhost:9000` if the server is up!

2. Deploy configuration

   ```
    SYNAPSE\_HOME/bin/synapse.bat -sample [EXAMPLE NUMBER]
   ```

3. Executing the sample client
   This configuration does vary. Check the example catalog or the section 4.6.

   But if there is no loadbalancer the following ant script will work:

   ```
   \{SYNAPSE_HOME\}/samples/axis2Client/ant stockquote
       -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
       -Dtrpurl=http://localhost:8280 -Dmode=quote -Dsymbol=IBM
   ```

A.3 LoadBalancer Example

The loadbalancer example[2] needs special configuration. The ESB needs 3 instances of the sample Axis2 server:

```
./axis2server.sh -http 9001 -https 9005 -name MyServer1
./axis2server.sh -http 9002 -https 9006 -name MyServer2
./axis2server.sh -http 9003 -https 9007 -name MyServer3
```

The Example number is 52.

The client can be invoked with:

```
 ant loadbalancefailover -Di=100
```

[2] http://synapse.apache.org/Synapse_Samples.html#Sample52

Bibliography

[BBKL13] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*, pp. 0–9. IEEE Computer Society Conference Publishing Services, 2013. (Cited on pages 11, 12, 18, 21 and 22)

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(03), 2012. doi:10.1109/MIC.2012.43. (Cited on pages 18 and 87)

[BFL^{+}12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. Formalizing the Cloud through Enterprise Topology Graphs. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 742–749. 2012. doi:10.1109/CLOUD.2012.143. (Cited on pages 12 and 18)

[BLNS12] T. Binz, F. Leymann, A. Nowak, D. Schumm. Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pp. 61–70. 2012. doi:10.1109/EDOC.2012.17. (Cited on page 18)

[Cha09] D. A. Chappell. *Enterprise service bus*. O'Reilly media, 2009. (Cited on pages 17, 24 and 51)

[Erd09] H. Erdogmus. Cloud Computing: Does Nirvana Hide behind the Nebula? *IEEE Software*, 26(2):4–6, 2009. doi:10.1109/ms.2009.31. URL http://dx.doi.org/10.1109/ms.2009.31. (Cited on page 15)

[ES08] C. Emmersberger, F. Springer. *Event Driven Business Process Management taking the Example of Deutsche Post AG - An evaluation of the Approach of Oracle and the SOPERA Open Source SOA Framework*. Ph.D. thesis, Fachhochschule Regensburg, Regensburg, 2008. URL http://epub.uni-regensburg.de/28590/. (Cited on page 11)

[Fre] P. Fremantle. Paul Fremantle's Blog - Synapse and WSO2 ESB myths. http://pzf.fremantle.org/2012/09/synapse-and-wso2-esb-myths.html. (Cited on page 53)

[HW02] G. Hohpe, B. Woolf. Enterprise Integration Patterns. In *9th Conference on Pattern Language of Programs*. 2002. (Cited on pages 19, 28, 29, 30, 31 and 63)

[IA10] C. Ibsen, J. Anstey. *Camel in action*. Manning Publications Co., 2010. (Cited on pages 20 and 23)

[Jet12] Jetty. Jetty The Definitive Reference. http://www.eclipse.org/jetty/documentation, 2012. (Cited on page 35)

[Kre12] J. Krein. *Framework for Application Topology Discovery to enable Migration of Business Processes to the Cloud*. Master's thesis, Universität Stuttgart, 2012. (Cited on page 88)

[MG09] P. Mell, T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology (NIST)*, 2009. (Cited on page 15)

[MLB$^+$11] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, A. Ghalsasi. Cloud computing - The business perspective. *Decis. Support Syst.*, 51(1):176–189, 2011. (Cited on page 16)

[Ora] Oracle Corporation. Java Architecture for XML Binding (JAXB). http://www.oracle.com/technetwork/articles/javase/index-140168.html. (Cited on page 71)

[ORA11] ORACLE Corporation. *Java Management Extensions*. ORACLE Corporation, java se 6 edition, 2011. (Cited on page 22)

[Ort07] S. Ortiz. Getting on Board the Enterprise Service Bus. *Computer*, 40(4):15–17, 2007. doi:10.1109/MC.2007.127. (Cited on page 11)

[PTDL07] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007. doi:10.1109/MC.2007.400. (Cited on page 11)

[RCL09] B. P. Rimal, E. Choi, I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, NCM '09, pp. 44–51. IEEE Computer Society, Washington, DC, USA, 2009. doi:10.1109/NCM.2009.218. URL http://dx.doi.org/10.1109/NCM.2009.218. (Cited on page 16)

[Red] Red Hat Inc. Fuse ESB Enterprise Data Sheet. http://fusesource.com/collateral/172. (Cited on page 34)

[Red12] Red Hat Inc. Hermes Case Study - Hermes Uses FuseSource for UKs Largest Retail Network. http://fusesource.com/collateral/104/, 2012. (Cited on pages 11 and 34)

[Roe12] P. Roehrig. New Market Pressures Will Drive Next-Generation IT Services Outsourcing. *Forrester Research, Inc. (2009)*, 2012. (Cited on page 11)

[Sea02] C. Seaman. The information gathering strategies of software maintainers. In *Software Maintenance, 2002. Proceedings. International Conference on*, pp. 141–149. 2002. doi:10.1109/ICSM.2002.1167761. (Cited on page 21)

[SL09] T. Scheibler, F. Leymann. From Modelling to Execution of Enterprise Integration Scenarios: The GENIUS Tool. In K. David, K. Geihs, editors, *KiVS*, Informatik Aktuell, pp. 241–252. Springer, 2009. URL http://dblp.uni-trier.de/db/conf/kivs/kivs2009.html#ScheiblerL09. (Cited on page 21)

[Spr13] SpringSource. *Spring Framework Reference Documentation*, 2013. (Cited on page 36)

[SV96] M. Shreedhar, G. Varghese. Efficient fair queuing using deficit round-robin. *Networking, IEEE/ACM Transactions on*, 4(3):375–385, 1996. doi:10.1109/90.502236. (Cited on page 38)

[TAW03] L. Titchkosky, M. Arlitt, C. Williamson. A performance comparison of dynamic Web technologies. *SIGMETRICS Perform. Eval. Rev.*, 31(3):2–11, 2003. doi:10.1145/974036.974037. URL http://doi.acm.org/10.1145/974036.974037. (Cited on page 36)

[Thea] The Apache Software Foundation. ActiveMQ. http://activemq.apache.org/. (Cited on page 34)

[Theb] The Apache Software Foundation. Apache CFX. http://cxf.apache.org/. Accessed: 2013-09-07. (Cited on page 34)

[Thec] The Apache Software Foundation. Apache ODE. http://ode.apache.org/. (Cited on page 34)

[Thed] The Apache Software Foundation. Apache Synapse. http://synapse.apache.org/. (Cited on pages 20, 53 and 55)

[Thee] The Apache Software Foundation. Apache Synapse Documentation. http://synapse.apache.org/docs_index.html. (Cited on pages 20, 51, 54 and 65)

[Thef] The Apache Software Foundation. Apache Synapse Samples Catalog. http://synapse.apache.org/userguide/samples.html. (Cited on page 66)

[Theg] The Apache Software Foundation. Camel Direct. http://camel.apache.org/direct.html. (Cited on page 34)

[Theh] The Apache Software Foundation. Major Retail Pharmacy Chain - Builds Warehouse Management System with Fuse ESB. http://fusesource.com/collateral/17/. (Cited on page 35)

[Thei] The Apache Software Foundation. ServiceMix. http://servicemix.apache.org/. (Cited on page 33)

[The13] The Apache Software Foundation. *Camel Manual*. Apache Software Foundation, version 2.11.0 edition, 2013. (Cited on pages 20, 23, 24, 27, 32, 33, 35 and 44)

[Top11] Topology and Orchestration Specification for Cloud Applications (TOSCA). OASIS specification. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`, 2011. (Cited on page 18)

[VZ09] J. Voas, J. Zhang. Cloud Computing: New Wine or Just a New Bottle? *IT Professional*, 11(2):15–17, 2009. doi:\url{http://doi.ieeecomputersociety.org/10.1109/MITP.2009.23}. (Cited on page 15)

[WCB01] M. Welsh, D. Culler, E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001. doi:10.1145/502059.502057. URL `http://doi.acm.org/10.1145/502059.502057`. (Cited on page 36)

[WSO12a] WSO2 Inc. eBay uses 100% open source WSO2 ESB to process more than 1 billion transactions per day. http://wso2.com/download/wso2-ebay-case-study.pdf, 2012. (Cited on page 65)

[WSO12b] WSO2 Inc. ESO2 Middleware ensures Alfa-Bank a promising future in SOA. `http://wso2.com/download/wso2-alfa-bank-case-study.pdf`, 2012. (Cited on page 65)

[WSO13] WSO2 Inc. *Enterprise Service Bus Documentation*. WSO2 Inc., version 4.7.0 edition, 2005-2013. (Cited on pages 57 and 65)

All links were last followed on October 14, 2013.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Andre Grund)