Digital Business Ecosystem

Contract nº 507953

Workpackage 17: Composer
Deliverable 17.4: Automatic Composer

Project funded by the European Community under the “Information Society Technology” Program

Contract Number: 507953
Project Acronym: DBE
Title: Digital Business Ecosystem

Deliverable Nº: 17.4
Due Dates: 10/2006
Delivery Date: 10/2006

Short Description: This deliverable describes the feature of dynamic service selection in Service-Oriented Architectures and its integration into the Servent. A detailed analysis of the reliability levels are provided with regards to different service failure rates and the ability to mask those failures transparently with our approach.

Author: Trinity College Dublin (TCD)
Partners Contributed:
Made Available To: Public

Versioning

Version	Date	Author, Organisation	Description
0.1	January/ February	Lotte Nickel, and René Meier, TCD	Background, Introduction
0.2	March/ April	Lotte Nickel, and René Meier, TCD	Design, Implementation, Evaluation
0.3	April	Lotte Nickel and René Meier, TCD	Review Cycle
0.4	June	Dominik Dahlem, TCD	Re-format
0.5	July	Dominik Dahlem, TCD	Style and Consistency
1.0	September/ October	Dominik Dahlem, Lotte Nickel, and René Meier, TCD	Submission to Reviewers
1.1	1st of November	Dominik Dahlem, Lotte Nickel, and René Meier, TCD	Final version
1.2	14th of November	Dominik Dahlem	Modifications of the header/footer styles

Quality Check:
1st Internal Reviewer: Pierfranco Ferronato, Soluta
2nd Internal Reviewer: Miguel Vidal, SUN

Contents

List of Figures

List of Listings

List of Tables

Executive Summary

Web services were introduced to support application-to-application communication over the Internet. These services provide standardised interfaces and are ignorant towards protocols in order to allow services from a variety of solution providers to interact. Various approaches to Web service composition have been proposed to facilitate the interaction between a set of Web services fulfilling a complex goal. These service compositions accept invocations from clients, coordinate the invocations of the necessary Web services and return the results to the clients. The Digital Business Ecosystem (DBE) project specifies an architecture for enabling small-to-medium sized enterprises (SMEs) to define, provide and consume Internet-based business services, such as an online booking service. It enables service providers to create and advertise business services so that service consumers can discover and use them.

In this scenario, Web service compositions suffer from two major problems: they depend on the availability of the individual Web services and their correct behaviour, and they lack the flexibility to replace a failed Web service with a redundant alternative. In a world of SMEs, the availability of services cannot be guaranteed for a long period of time and the DBE architecture needs to address these issues. If one of these services is not available or fails during execution, the architecture needs to be able to dynamically switch to alternative Web services that provide equivalent functionality. These problems escalate with the introduction of service compositions because the replacement of a service composition with an alternative composition is a not well understood issue with few proposed solutions.

In order to address this issue, we propose a coordination service composition architecture that offers service instance and service composition failure resilience through the ranking of available service compositions and the coordination of these compositions. This is achieved through the introduction and modification of a number of specific components in the DBE architecture.

This deliverable explains the design and implementation of the coordination service composition architecture and presents an evaluation of the architecture against a set of scenarios.

Chapter 1

Introduction

This deliverable presents an architecture for coordinating service compositions in a *Digital Business Ecosystem* (DBE). It supports the provision of more reliable and failure resilient service compositions to service consumers than non-coordinated service compositions. This chapter motivates the work, introduces the concepts of the DBE and service compositions and presents the scope of this deliverable.

1.1 Motivation

1.1.1 Digital Business Ecosystem

As the importance of using Internet-based business services increases steadily, more and more *small-to-medium sized enterprises* (SMEs) start to realise the advantages that the Internet can offer them. The Internet can be considered as one of the least expensive ways to market products globally and all sites on the Internet are considered to be equal [54], enabling SMEs to compete with larger enterprises. However, SMEs suffer from disadvantages in connection with their size when it comes to human resources and financing technical equipment, software and expertise. Often, these disadvantages undermine their ability to compete with larger enterprises [14].

As SMEs become increasingly important for their shares in employment and business turnover to Europe's economy [14], the *European Union* (EU) initiates research projects such as the DBE [18] project to strengthen and support the SME industry. The DBE is an open-source distributed environment, in which Internet-based business services can be designed, provided and consumed. The DBE enables service providers to create and advertise business services so that service consumers can discover and use them. The target-clientele of the DBE are SMEs that might have low *Information and Communication Technology* (ICT) access, such as non-permanent or low-bandwidth Internet connections.

This has the following implications on the design and structure of services for the DBE. Firstly, the DBE cannot assume that services remain available over a long period of time. Services might become temporarily unavailable due to provider restrictions, for example, services might be made available during business hours only, or due to network or host failures. As indicated by Papazoglou [44], service failures are an important aspect to be considered especially for SMEs that compete with larger enterprises. The issue of service availability increases with the usage of compositions of

services, as service compositions depend on a number of other services during their execution. A single service failure can lead to a complete failure of a service composition, lessening the reputation of a service provider and diminishing the confidence of service consumers to the extent of putting a service provider out of the business market. Therefore, from a service provider's perspective, service reliability and failure resilience are two of the most important aspects that must be considered.

> "Any downtime of key e-business systems has a negative impact on business to the extent of throwing you out of the market."[44]

Secondly, in order to compete with larger enterprises, SMEs must be able to use the technology provided within the DBE easily and at a low cost. Therefore, the DBE project will be released as an open-source and standards-based environment, which implies that only open-source technologies may be used and existing standards must be used (extended) where possible. Ease of use is another important characteristic for the design of services in the DBE as it cannot be assumed that SMEs have a deep technical knowledge.

Finally, the DBE is based on a peer-to-peer environment to distribute core services of the DBE, such as services that store or replicate data, on stable peers that are well-connected, have high bandwidth and processing power [48]. This approach enables SMEs with a low ICT access to participate in activities of the DBE as they are not required to run core services. However, SMEs that want to offer services to consumers must either run a set of DBE specific services locally on their network, which might pose problems to SMEs with a low ICT access, or find another service provider that hosts their service.

1.1.2 Composing Web Services

In recent years, the area of Web service composition attracted much interest from industry as well as from academia. This interest originates from the increased usage of business services over the Internet, which opens up new vistas for long-running business-to-business interactions and transactions [55]. Rather then writing new and more complex services from scratch, Web service composition aims at combining existing services into new services. This approach follows one of the main concepts of software development:

> "Good programmers know what to write. Great ones know what to rewrite (and reuse)." [47]

Web services support simple interactions, using standard messages and protocols and are essentially stateless in between operations, except for the state between sending and receiving a message [55]. In order to support long-living business processes and transactions, it is essential to have a process model that specifies the order in which operations execute and keeps the state to determine the next step in a business process [55]. This requirement triggered the evolvement of service composition languages such as the *Business Process Execution Language for Web Services* (BPEL) [5], the *Business Process Modelling Language* (BPML) [33], *Web Services for Business Process Design* (XLANG) [53] and the *Web Services Flow Language* (WSFL) [13]. BPEL, itself a combination of its predecessors XLANG and WSFL, is likely to emerge as a widely accepted standard for defining executable business processes in the industry and is currently in the process of being standardised. As illustrated in Figure 1.1, BPEL adds

a service composition layer on top of Web services standards such as *Web Services Description Language* (WSDL) [8] and *Simple Object Access Protocol* (SOAP) [55], while offering a WSDL interface to service consumers.

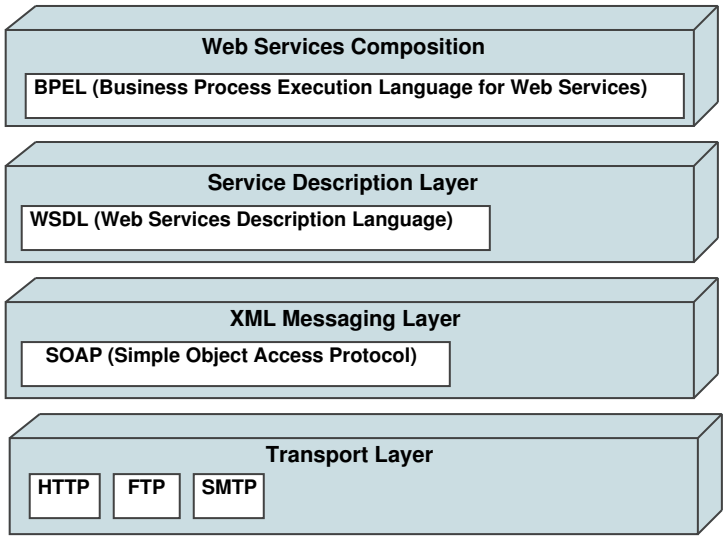


Figure 1.1: Overview of Web services technology

BPEL defines a process-centric model for the formal specification of the behaviour of business processes based on the interaction of an executable process with its partners [50], exchanged messages, fault and compensation handling mechanisms [63]. A BPEL process both provides and uses more than one WSDL service [55], which are also called partners. A WSDL service is defined by abstract and concrete definition parts [41]. The abstract part describes ports that offer operations (*service types*) while the concrete part defines binding and endpoint information of a service (*service instance*). In order to provide a composite service, a BPEL process specifies the order in which the operations execute [55]. Activities are used to structure a BPEL process and to determine the next activity that must be executed. The state of a BPEL process is stored in variables, which are essentially containers for storing values that can be accessed and modified using XPath [62] expressions.

A BPEL process becomes executable when it is deployed on a BPEL workflow engine, such as the ActiveBPEL engine [2]. Although a BPEL process knows only of partners and is unaware of the underlying dependency on concrete WSDL bindings and endpoints, these are required by workflow engines at deployment time [10]. A workflow engine exposes a deployed BPEL process as a concrete WSDL interface to service consumers.

BPEL defines the combination of `faultHandlers` and a `compensationHandler` to catch failures and compensate finished activities within a transaction scope. However, this failure handling assures that a BPEL service composition can finish processing gracefully, compensating or cancelling any transactions. Rather than returning a failure message to a service consumer, alternative services for the failed service composition should either be offered to a service consumer or executed automatically where possible.

1.1.3 Coordinating Web Service Compositions

Service compositions can be composed of several services, which might be deployed locally on the same machine or externally anywhere on the Internet. If not all used services are available to a service composition at run-time it will fail. However, the availability of services in the DBE cannot be guaranteed at all times. In order to provide service compositions that are failure resilient and offer high service composition reliability in the DBE, it is essential to enable automatic fail-over and execution of alternative service compositions at run-time. The process of deploying, executing and replacing alternative service compositions is called coordinating Web service compositions. A list of requirements can be compiled that dictate the design of an architecture for coordinating service compositions in the DBE. The requirements cover three areas:

1. The architecture must be open-source and standards-based because the DBE is an open-source environment.

2. If more than one service composition offering equivalent functionality is available, the architecture must delegate a service consumer's requests to the highest ranked service composition based on Quality of Service (QoS) information for the service compositions.

3. The architecture must offer advanced failure handling and switch automatically to alternative service compositions. This is one of the main requirements as services are likely to fail within the DBE, leading to the failure of a service composition otherwise.

4. Ease of use for consumers is another important requirement in the sense that a consumer remains unaware of the architecture.

1.2 Contributions

Within the context of this deliverable, an architecture for the coordination of service compositions in the DBE was designed and implemented that enables service providers to achieve higher service composition reliability, failure resilience and ease of use for service consumers compared to individual service compositions. A service composition approach in the DBE must be open-source and standards-based. At the same time, the issue of constantly changing service availability within a DBE must be taken into account when designing service compositions.

The main contribution of this deliverable is the design and implementation of a *coordination architecture*. A service provider may supply several service compositions that offer equivalent functionality to service consumers but are structured differently and use different services within their composition. Instead of deploying and advertising each service composition individually, the coordination architecture enables a service provider to make a set of service compositions with a single entry point available to service consumers. This structure creates a redundancy of service compositions that can be used to ranked the service compositions and choose the highest ranked one. Another advantage is that a failure in a selected service composition can be handled and another service composition can be chosen as a replacement.

The coordination architecture introduces the following mechanisms and features:

- Coordinator:
  One mechanism that the coordination architecture introduces is the *coordinator*. The coordinator is a service composition that coordinates other service compositions that are deployed locally on the same machine. All service compositions offer equivalent functionality while using different services within their service compositions. The coordinator acts as the interaction point for a service consumer. Upon receiving a request from a service consumer, the coordinator decides at runtime which service composition is the highest ranked one that will be used to execute the request. The service consumer's request is then delegated to the selected service composition, which starts executing the request. If the coordinator detects that the selected service composition has failed, the failure is caught and handled. The coordinator determines the next highest ranked service composition and redirects the service consumer's request to this service composition.

- Ranking Service:
  A feature that the coordination architecture introduces is the *Ranking Service*. The Ranking Service has access to QoS information for services within the DBE, such as failure and performance history of a service that can be used to create a ranking of available and equivalent services. The Ranking Service is used for two tasks. Firstly, the coordinator instructs the Ranking Service to retrieve the highest ranked service out of a given set of services. Secondly, the Ranking Service is used to update the QoS information about a service for example after being informed about a service failure by the coordinator.

- Failure Handling:
  The coordination architecture offers a two-step failure handling mechanism. Services within the DBE are identified by a unique identifier and their endpoint information is retrieved dynamically at run-time from a distributed lookup service. If more then one service instance implements the same service type it can reuse this identifier. Therefore, at run-time there might be several service instances available for the same service type, which enable a fail-over to another service instance. If this failure handling fails, the coordination architecture defines the coordinator for providing a second failure handling mechanism that enables fail-over to another service composition. The usage of redundant and replaceable service compositions increases the chance that one service composition will finish executing a service consumer's request.

An implementation of this architecture for coordinating BPEL service compositions has been evaluated using a coordinator that coordinates five BPEL service compositions, which all depend on three individual services. The coordinator and each individual BPEL service composition are tested for their failure resilience and service composition reliability at different failure rates (5%, 10%, 20% and 40%) for the individual services. Five clients are sending request to a tested composition concurrently in order to test a compositions realistically. This evaluation demonstrates that the coodination architecture achieves a higher service composition reliability and failure resilience, while leaving consumers unaware of the coordination architecture.

1.3 Scope

The focus of this deliverable is to present an architecture for the coordination of BPEL service compositions in a DBE. Security and permission issues are important for an environment such as the DBE. However, they are considered to be beyond the scope of this deliverable.

1.4 Roadmap

The remainder of this deliverable is structured as follows: Chapter 2 introduces the terminology and characteristics of service compositions and subsequently reviews work related to this deliverable. Chapter 3 describes the design of the architecture for coordinating BPEL service compositions in a DBE. Chapter 4 presents a prototypical implementation of the architecture. In chapter 5, an evaluation of the prototype is presented. Finally, chapter 6 concludes this deliverable by summarising the presented work and outlining issues that remain open for future work.

Chapter 2

Background and Related Work

This chapter establishes the foundation of the deliverable. It introduces background work on the DBE project and Web service compositions. The architecture of the DBE is presented followed by the introduction to deployment and execution of services. The service composition problem and current solutions such as the BPEL and OWL-S. Related work with an emphasis on BPEL is discussed and a comparison that identifies limitations within existing work is provided.

2.1 Digital Business Ecosystem (DBE)

This section gives an overview of the DBE project, its architecture and introduces important components that will be reused or extended in the design chapter for coordinating BPEL compositions. The following subsection presents an overview of the DBE architecture. The second and third subsections introduce the deployment and execution of DBE services respectively.

2.1.1 DBE Overview

SMEs become increasingly important for their shares in employment and turnover to European countries [14]. The problem of SMEs to compete with larger enterprises due to financial and technical disadvantages [54] created the demand within the European Union to enable SMEs competing at high-level no matter how small or remote their business is. Therefore, the European Commission's 6th Framework Programme for research and development in Information Society Technologies supported work on a DBE, the DBE project. The DBE provides an open-source environment for Internet-based business services with the target clientele of SMEs that might only have low a Information and Communication Technology (ICT) access. The DBE enables the design, composition, advertisement and consumptions of business services.

2.1.2 DBE Architecture

The DBE architecture consists of three distinct environments: the Service Factory (SF), the Execution Environment (ExE) and the Evolutionary Environment (EvE) [32]. These environments and their core components are depicted in Figure 2.1.

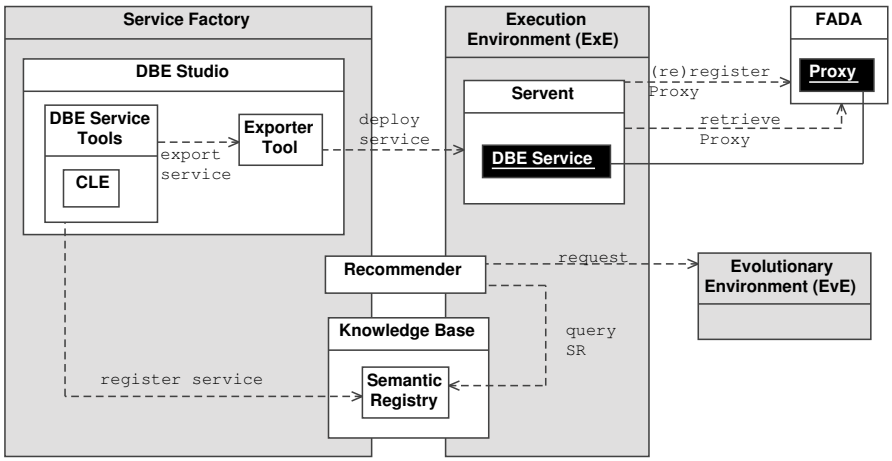


Figure 2.1: DBE architecture

The SF is used for the creation and advertisement of business models, service models and ontologies. The ExE is the deployment and execution environment for services in the DBE with the Servent as the application server [20]. The EvE is an long-term experimental environment that is used to store service usage and performance history from the SF and ExE.

A main component of the SF is the DBE Studio [19]. The DBE Studio is an open-source development environment for the DBE that provides a collection of DBE service tools and an Exporter Tool (ET). The DBE service tools can be used to analyse and define service models and create DBE services. A DBE service is deployed and published to the ExE using the ET. Part of the DBE service tools is the Composition Language Editor (CLE). The CLE provides a set of editors and wizards that can for example be used to discover and select available services from the Semantic Registry for inclusion in the BPEL service composition. If several equivalent services are available, the Recommender can be used to give recommendations of services [37]. The Semantic Registry is part of the Knowledge Base that is used to store business models, service models and ontologies [29].

In the ExE, services are deployed to a Servent instance. The Servent registers a service's Proxy with FADA *(*Federated Advanced Directory Architecture*)* [52]. FADA is a distributed lookup service currently used in the ExE. The deployment and execution of DBE services is explained in more detail in the following sections. The Servent also provides a set of structural features including authentication, authorisation, accounting, privacy and transactions but these are not relevant to this deliverable.

2.1.3 Deployment and Advertisement of Services

Figure 2.2 describes the creation, deployment and advertisement of a DBE service. A service provider uses the tools of the DBE Studio to create a DBE service. A new DBE service is registered in the Semantic Registry. On registering a DBE service in the Semantic Registry for the first time, a service provider receives a unique identifier (SMID) from the Semantic Registry. The SMID is used by a service provider to register the service in FADA.

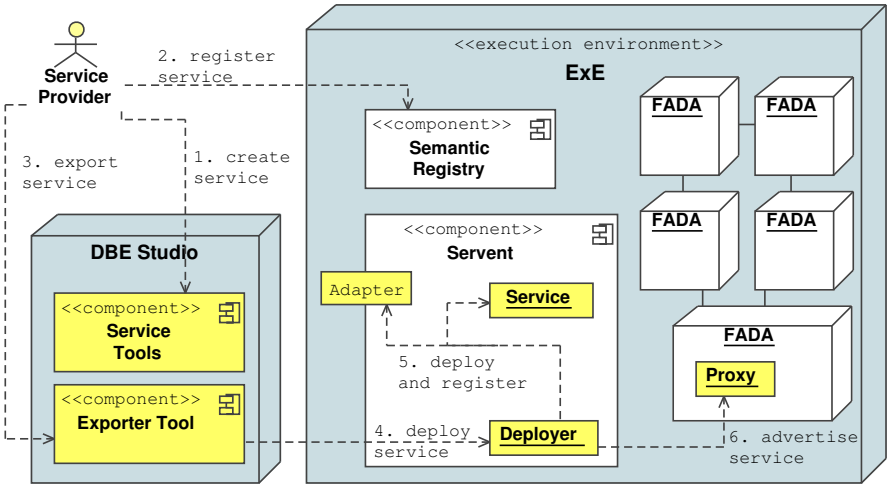


Figure 2.2: Deployment and advertisement of a service in DBE

A service provider uses the ET to deploy a DBE service to a Servent instance. A DBE service consists of a service implementation and a deployment descriptor. A DBE service must implement an adapter interface that the Servent uses to deploy and undeploy a service, the `org.dbe.servent.Adapter` interface shown in Listing 2.1. The Servent calls the `init()` method at deploy-time and the `destroy()` method at undeploy-time.

Listing 2.1: Adapter interface

```
1 public interface org.dbe.servent.Adapter
2 {
3   void init (ServiceContext context);
4   void destroy ();
5 }
```

The deployment descriptor contains information about the service such as its SMID and the service adapter implementation. A minimal version of a deployment descriptor for a DBE service is depicted in Listing 2.2:

Listing 2.2: Content of a minimal deployment descriptor

```
1 <deploymentProperties>
2    <serviceInfo>
3       <serviceName>MyName</serviceName>
4       <smid>my–SMID</smid>
5       <adapterClass>org.dbe.service.ServiceAdapter</
            adapterClass>
6    </serviceInfo>
7 </deploymentProperties>
```

The ET packages the service implementation in conjunction with the deployment descriptor into a DBE archive (DAR) and hands the DAR over to the Servent in the

ExE. The Deployer component of a Servent instance deploys a DBE service locally and registers it with the Adapter. The Adapter stores a DBE service implementation with a key in a repository. The key is a combination of the Servent URL `http://hostname:2727/` and an identifier for the service. The key is used by the Adapter to correlate incoming requests with the correct DBE service instance. The Deployer then creates a Proxy for the DBE service and advertises it to the local FADA instance of the Servent. The advertisement of Proxies must be renewed in regular intervals by the Servent otherwise the Proxy will be removed from FADA.

A Proxy is a Java object that encapsulates the configuration information for a service. Within this information the port-specific data, mainly the current endpoint address and invocation protocol for a service, can be found and used to interact with a service. Proxies are identified via the SMID of a DBE service in FADA. A more detailed description of Proxy objects is available in [15].

2.1.4 Discovery and Consumption of Services

Figure 2.3 describes the discovery and consumption of a DBE service, shown in . A service consumer browses the Semantic Registry for available service types and retrieves the SMID of a suitable DBE service. In order to find the Proxy for the required service, the consumer contacts a FADA node with a request that contains the SMID of the required service. Upon successful discovery, the consumer downloads the Proxy object and invokes on it. The Proxy contains all required information, such as a GUI, that a consumer needs to use the corresponding service.

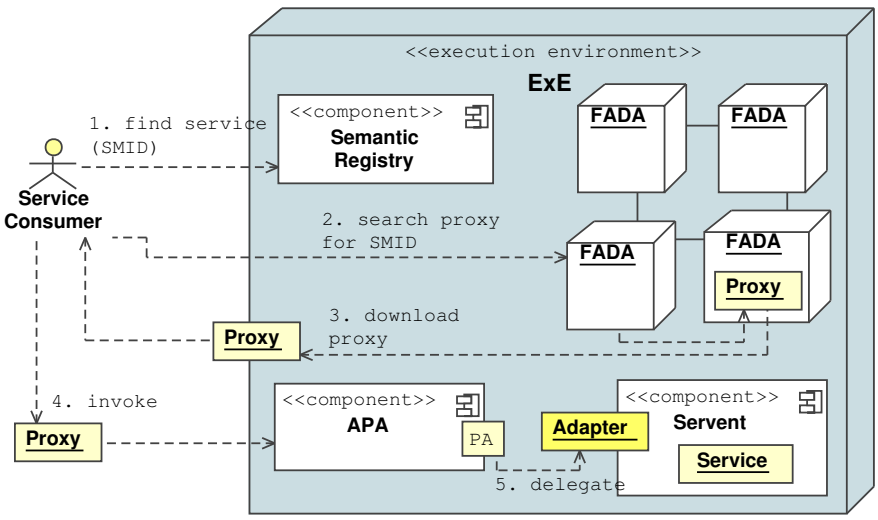


Figure 2.3: Discovery and consumption of a service in DBE

The service consumer invokes on the downloaded Proxy and the invocation is delegated via the *Abstract Protocol Adaptor* (APA) to the Servent. As presented in [15], the APA provides a dynamic invocation interface that promotes loose coupling between clients and the protocol used to access a services. A Proxy object configures the APA at invocation time with the necessary endpoint and the required invocation protocol.

The APA is ignorant of protocols and simply delegates the invocation to the respective concrete protocol adaptor (PA) at runtime. The PA for the Servent uses object serialisation over HTTP to delegate the invocation to the Servent's Adapter. The Adapter correlates a received request with the correct DBE service and delegates the invocation to an instance of this service. The correlation is performed with the help of a repository that uses a key to DBE service implementation mapping. This enables the Adapter to retrieve the correct DBE service implementation. The DBE service instance finally executes the request.

2.2 Web Service Composition

A fundamental characteristic of large computer networks, such as the Internet and corporate intranets, is that they are heterogeneous [58]. The demand for a standardised middleware that simplifies connecting computers with different hardware and software platforms, from different eras of computer history, different vendors and designed for different purposes resulted in middlewares such as CORBA [30]. However, CORBA is mainly used on the intranet and is based on client-server interactions. The Internet required a new approach for a middleware that enables application-to-application interaction rather than client-server integration [59, 51]. In order to solve this problem, Web services provide a systematic and extensible framework built on top of existing Web protocols and based on open XML standards [11, 60]. Web services support simple interactions, using standard messages and protocols [61]. However, they are essentially stateless in between operations, except for the state between sending and receiving a message [55]. In order to support long-living business processes and transactions, it is essential to have a process model that specifies the order in which operations execute and keeps the state to determine the next step in a business process [55]. This requirement triggered the evolvement of service composition languages. The research on Web service composition is extensive and various service composition languages, frameworks and architectures have been defined [17, 40]. The service composition problem can be partitioned into two main problem areas [3]:

1. Creating a service composition:
   A service provider must first decide on the functionality that a service composition will offer to service consumers. The next step is to discover available service types that provide parts of this functionality. The service provider then specifies the services that are used within the composition, and the order in which the services must be executed to achieve the desired functionality. Different service description languages, annotating services with semantics or not, and different service composition languages are important aspects at creation time of a service composition.

2. Executing a service composition:
   The execution of a service composition introduces new aspects to the service composition problem. The service composition must discover service instances for used service types within a composition. A service failure at execution time can lead to a complete failure of a service composition. Service discovery, extensive failure handling and transaction management are some important aspects at execution time.

Research reveals that mainly two different approaches have been taken to standardise and compose Web services [3, 7]. The business world focuses on practical and exe-

cutable solutions, seeing Web services as abstract, standardised interfaces to business processes. Web services are described using WSDL, are composed with BPEL and invocations are delivered via the SOAP protocol. WSDL only allows the syntactic specification of messages that enter or leave a web service application [7] while BPEL describes the order in which such messages are to be exchanged [7, 50]. Therefore, this approach leads to the definition of static BPEL processes that must be declared manually [7, 50]. The Semantic Web Community approaches the web service composition problem from the perspective of enhancing Web services with semantic descriptions using ontologies that are computer-interpretable. They envision long-term goals like automatic discovery, composition, and execution of services based on semantic Web services [7, 38].

BPEL has been a widely accepted as a solution for defining executable business processes in the industry and is currently in the process of being standardised. On the other hand OWL-S is used in most semantic approaches to Web service composition. The following two sections provide an overview of the two approaches with the emphasis on BPEL as it is used within this deliverable for defining service compositions.

2.2.1 BPEL

BPEL [5, 63] emerges as the accepted standard language for defining executable business processes. Coauthored by BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, the latest BPEL version 1.1 that was released in May 2003 received industry-wide support [55].

BPEL defines a process-centric model for the formal specification of the behaviour of business processes based on the interactions of an executable process with its partners [50], exchanged messages, fault and exception handling mechanisms [63]. Figure 2.4 summarises the important features of the BPEL language.

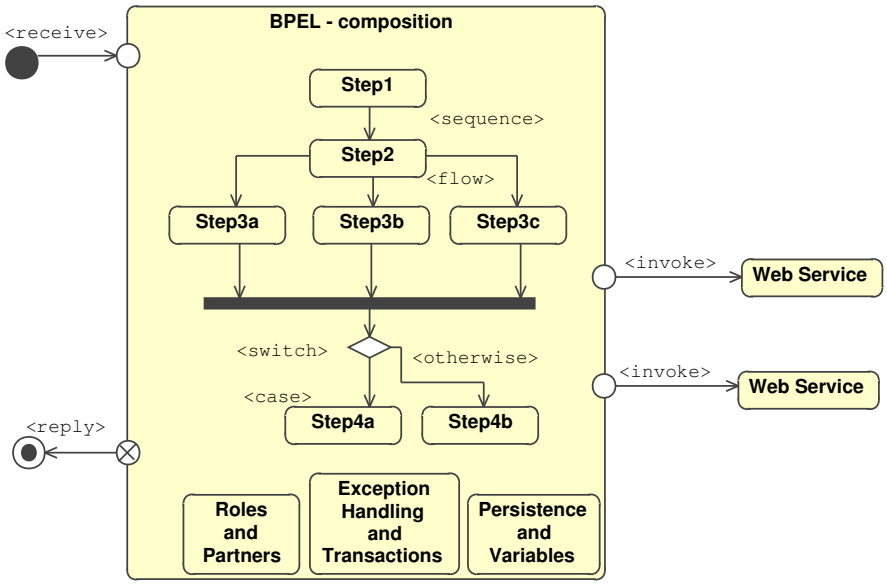


Figure 2.4: BPEL process overview

The basic functionality of a BPEL service composition is to receive requests from service consumers, invoke a number of Web services and issue a reply to the consumer. In order to understand the details of these actions a number of structures such as WSDL interfaces, `partnerLinks` and `variables` are needed. We will describe these in the following sections and then explain the individual activities: `receive`, `invoke` and `reply`.

2.2.1.1 WSDL Interface

A WSDL interface can be separated into an abstract (service type) and a concrete part (service instance) [11, 41]. The abstract service type information is defined by `message` and `portType` elements while the concrete service instance information is defined by `service` and `binding` elements. Listing 2.3 depicts a sample WSDL interface.

Listing 2.3: WSDL interface

```
1  <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       targetNamespace="this-ns" xmlns:ns="this-ns">
2    <message name="bookingFault">
3      <part name="failureReason" type="xsd:string"/>
4    </message>
5    <message name="clientRequest">
6      <part name="name" type="xsd:string" />
7      <part name="dates" type="xsd:string" />
8    </message>
9    <message name="clientResponse">
10     <part name="reservationNo" type="xsd:string" />
11     <part name="successful" type="xsd:boolean" />
12   </message>
13   <portType name="ClientPT">
14     <operation name="booking" parameterOrder="bookingData">
15       <input name="clientRequest" message="ns:clientRequest"
             />
16       <output name="clientResponse" message="
             ns:clientResponse"/>
17       <fault name="errorDuringBooking" message="
             ns:bookingFault"/>
18     </operation>
19   </portType>
20   <binding name="ClientSoapBinding" type="ns:ClientPT">
21     <soap:binding style="rpc" transport="http://schemas.
         xmlsoap.org/soap/http"/>
22     <operation name="booking">
23       <soap:operation/>
24       <input>
25         <soap:body use="encoded" encodingStyle="http://
             schemas.xmlsoap.org/soap/encoding/" namespace="ns
             "/>
26       </input>
27       <output>
```

```
28          <soap:body use="encoded" encodingStyle="http://
                schemas.xmlsoap.org/soap/encoding/" namespace="ns
                "/>
29        </output>
30      </operation>
31    </binding>
32    <service name="ClientService">
33      <port name="Client" binding="ns:ClientSoapBinding">
34        <soap:address location="http://host:2728/active-bpel/
              services/Client"/>
35      </port>
36    </service>
37  </definitions>
```

The abstract service information contains the description of messages that can be exchanged (`message`). A `portType` defines an endpoint of a Web services that supports a collections of operations. Each `operation` can define input, output and fault messages. A consumer contacts a `portType` and invokes an `operation`.

The concrete service information contains what communication protocol a consumer must use to interact with the Web service (SOAP in this example), how information is sent (RPC) and the endpoint of the Web service (`http://host:2728/active-bpel/services/Client`).

2.2.1.2 PartnerLinks

The `partnerLink` element describes links to services that are invoked by the process as well as links to service consumers that may invoke the process [45]. A BPEL process may use multiple Web services each defined by a `partnerLink` element. A BPEL process exposes a WSDL interface to service consumers, such as is shown in Listing 2.4.

Listing 2.4: PartnerLinks in WSDL interface of BPEL process

```
1  <definitions>
2    <!-- ... -->
3    <plnk:partnerLinkType name="ClientLink">
4      <plnk:role name="Client">
5        <plnk:portType name="ns:ClientPT"/>
6      </plnk:role>
7    </plnk:partnerLinkType>
8    <plnk:partnerLinkType name="HotelLink">
9      <plnk:role name="Hotel">
10       <plnk:portType name="ns:HotelPT"/>
11     </plnk:role>
12   </plnk:partnerLinkType>
13 </definitions>
```

The example extends Listing 2.3. The two `partnerLink` elements represent the interaction of the process with a service consumer `Client` and a hotel service (see Listing 2.7 for hotel service). Within BPEL these elements are redefined as shown in Listing 2.5.

Listing 2.5: Example of the partnerLinks element

```
1 <partnerLinks>
2   <partnerLink name="Client" partnerLinkType="ns:ClientLink"
3       myRole="Client"/>
4   <partnerLink name="Hotel" partnerLinkType="ns:HotelLink"
5       partnerRole="Hotel"/>
6 </partnerLinks>
```

The first `partnerLink` defines the link of the WSDL interface that a BPEL process offers to service consumers (`myRole`). Lines 4-5 define the hotel service as a link to a `partnerLink` that might be invoked within the BPEL process, indicated by `partnerRole` .

2.2.1.3 Variables

A BPEL process stores and passes values in `variable` elements. A `variable` is essentially a container for storing values that can either have a simple structure such as `string` , `boolean` or `integer` or represent a WSDL message. The `variable` element has the subgroups `inputVariable` and `outputVariable` that are used for passing values to invocations and retrieving values respectively. Listing 2.6 illustrates how `variable` elements are defined within a BPEL process.

Listing 2.6: Defining variables in BPEL

```
1 <variables>
2   <variable messageType="ns:clientRequest" name="
        v_clientRequest"/>
3   <variable messageType="ns:clientResponse" name="
        v_clientResponse"/>
4   <variable messageType="ns:hotelRequest" name="
        v_hotelRequest"/>
5   <variable messageType="ns:hotelResponse" name="
        v_hotelResponse"/>
6 </variables>
```

The `variables` in Listing 2.6 refer to WSDL `messages` , the first two are defined in Listing 2.4 while the other are defined in the following Listing 2.7.

Listing 2.7: Defining example of WSDL messages and complex type for hotel service

```
1 <complexType name="BookingInformation">
2   <sequence>
3     <element name="name" type="xsd:string"/>
4   </sequence>
5   <sequence>
6     <element name="dates" type="xsd:string"/>
7   </sequence>
8 </complexType>
9 <message name="hotelRequest">
10   <part name="bookingData" type="ns:BookingInformation" />
11 </message>
12 <message name="hotelResponse">
```

```
13   <part name="reservationNo" type="xsd:string" />
14   <part name="successful" type="xsd:boolean" />
15 </message>
16 <portType name="HotelPT">
17   <operation name="bookHotel" parameterOrder="reservationNo
         successful">
18     <input name="hotelRequest" message="ns:hotelRequest"/>
19     <output name="hotelResponse" message="ns:hotelResponse"/>
20   </operation>
21 </portType>
```

2.2.1.4 Assigning Values to Variables

The specification of BPEL requires any implementation of BPEL to support the usage of XPath 1.0 as the expression language [5]. The primary purpose of XPath is to extract and modify parts of an XML document. XPath also provides basic facilities for manipulation of strings, numbers and booleans [62]. The usage of XPath within BPEL for assigning values to variables is shown in Listing 2.8

Listing 2.8: Assigning values to variables in BPEL

```
1 <assign name="assign-values-to-variables">
2   <copy>
3     <from part="name" variable="v_clientRequest"/>
4     <to part="bookingData" query="/bookingData/name" variable
          ="v_hotelRequest"/>
5   </copy>
6   <copy>
7     <from expression="true()"/>
8     <to part="successful" variable="v_clientResponse"/>
9   </copy>
10  <copy>
11    <from expression="concat('some-text:',
          bpws:getVariableData('v_hotelResponse', '
          reservationNo'))"/>
12    <to part="reservationNo" variable="v_clientResponse"/>
13  </copy>
14 </assign>
```

In the first assignment, the parameter `name` of the `v_clientRequest` variable in Listing 2.6 (linked to `clientRequest` WSDL message in Listing 2.4) is copied to the complex type `bookingData` of the `v_hotelRequest` variable in Listing 2.6 (linked to `hotelRequest` WSDL message in Listing 2.7). The second assignment uses an XPath expression to assign the boolean value `true` to the parameter `successful` of the `v_clientResponse` variable. Finally, a more complex XPath expression is used to concatenation text with the output of the `reservationNo` parameter in the `v_hotelResponse` variable and copy it to the `reservationNo` parameter in the `v_clientResponse` variable.

2.2.1.5 Interaction with Service Consumer and Web Services

Requests from service consumers are translated by a BPEL workflow engine to a `receive` activity, which is the entry point to a BPEL process. Upon an invocation of a `receive` activity, a BPEL process starts executing its process. The `receive` activity specifies the operation that a consumer has invoked. Any input values are kept in a variable. As Listing 2.9 illustrates, the `operation` and `portType` are the same as defined in Listing 2.4 while `variable` and `partnerLink` refer to Listing 2.6 and Listing 2.5, respectively.

Listing 2.9: Receive activity in BPEL

```
1 <receive name="receiving" createInstance="yes" operation="
    booking"
2     partnerLink="Client" portType="ns:ClientPT" variable="
        v_clientRequest"/>
```

`Invoke` activities are used to invoke operations of external services defined by `partnerLink` elements. As Listing 2.10 illustrates, the used `operation` and `portType` are the same as defined in Listing 2.7 while `inputVariable` and `outputVariable` refer to Listing 2.6 and `partnerLink` refers to Listing 2.5.

Listing 2.10: Invoke activity in BPEL

```
1 <invoke name="invoking" inputVariable="v_hotelRequest"
2     operation="bookHotel" outputVariable="v_hotelResponse"
3     partnerLink="Hotel" portType="ns:HotelPT"/>
```

Finally, a BPEL process uses the `reply` activity to complete execution and to return any values to a service consumer. As Listing 2.11 illustrates, the used `operation`, `portType` and `partnerLink` are the same as for the `receive` activity. The variable contains the reply values for a service consumer.

Listing 2.11: Reply activity in BPEL

```
1 <reply name="replying" operation="booking"
2     partnerLink="Client" portType="ns:ClientPT" variable="
        v_clientResponse"/>
```

2.2.1.6 Structuring and Controlling in a BPEL Process

The BPEL process is described in `sequence` and `flow` elements that define sequential and parallel execution of activities. Decision elements such as `switch` and `while` can be used to control the process and determine the next step depending on previous results. An example of a `switch` is given in Listing 2.12 that tests whether the value of the `v_clientResponse variable` is true. If so, the activity within `case` is executed. It is possible to use more than one `case` statement. The activity in `otherwise` is only executed if none of the `case` statements were true.

Listing 2.12: Switch activity in BPEL

```
<switch name="is-booked">
  <case condition="bpws:getVariableData('v_clientResponse', '
      successful')">
    <!-- do something -->
  </case>
  <case condition="bpws:getVariableData('v_hotelResponse', '
      successful')">
    <!-- do something else -->
  </case>
  <otherwise>
    <!-- if none of the other cases true, do this -->
  </otherwise>
</switch>
```

2.2.1.7 Failure Handling in BPEL

A failure in one of the used `partnerLink` elements results in partial or complete failure of a BPEL service composition. A complete failure occurs if a fundamental part of a BPEL service composition cannot be executed. Minor failures that can be neglected lead to a partial failure. The BPEL language offers simple failure handling within a composition so that errors during the BPEL workflow execution can be caught and compensated. The use of ACID transactions is limited to short periods of time and cannot support long-living business transactions [34]. BPEL introduces the concept of `scope`, `faultHandler` and `compensationHandler`, to delineate nested transactions within a BPEL process. If a failure occurs within a `scope`, this failure is caught and handled within a `faultHandler`. The `faultHandler` calls a `compensationHandler` to cancel any started transactions and compensate finished transactions within this `scope`.

A `faultHandler` element can contain several `catch` blocks that describe actions to be executed upon specific faults and one `catchAll` block that defines default actions upon any other faults. An example of a `scope`, a `faultHandler` and a `compensationHandler` is shown in Listing 2.13. If a failure occurs during calling `bookHotel`, the defined `faultHandler` will catch this failure and call the `compensationHandler` to handle it.

Listing 2.13: Standard FaultHandler in BPEL

```
<scope>
  <compensationHandler>
    <!-- compensate some activity -->
  </compensationHandler>
  <faultHandlers>
    <catch faultName="ns:HotelFault" faultVariable="
        HotelFault">
      <!-- handle the fault -->
    </catch>
  </faultHandlers>
  <invoke name="bookHotel" .../>
</scope>
```

Although `faultHandler` elements are usually used within a `scope`, a simple `faultHandler` exists for an `invoke` activity as illustrated in Listing 2.14:

Listing 2.14: FaultHandler for invoke activity in BPEL

```
<invoke name="bookHotel" ...>
  <catch faultName="ns:HotelFault" faultVariable="HotelFault"
      >
    <!-- handle HotelFault -->
  </catch>
  <catchAll>
    <!-- handle any other fault -->
  </catch>
</invoke>
```

2.2.1.8 The ActiveBPEL Engine

A BPEL process becomes executable when it is deployed on a BPEL workflow engine, such as the ActiveBPEL engine [2]. The ActiveBPEL engine is an open source implementation of a BPEL workflow engine. The ActiveBPEL organisation has implemented a pluggable workflow architecture in Java that is fully compliant with the BPEL specification, version 1.1 [5]. The engine runs on a servlet container and uses the Axis Web services container for communicating with other Web services and clients. It supports SOAP bindings for service invocations.

The ActiveBPEL engine is highly configurable via a configuration file. The configuration file is imported by the ActiveBPEL engine at startup and can be customised. Entries in the configuration file specify a number of factories, managers and handlers. An example for a handler is the `AeInvokeHandler` class that implements an `invoke` activity. All entries in the configuration file can be exchanged or customised without having to change any logic within the ActiveBPEL engine.

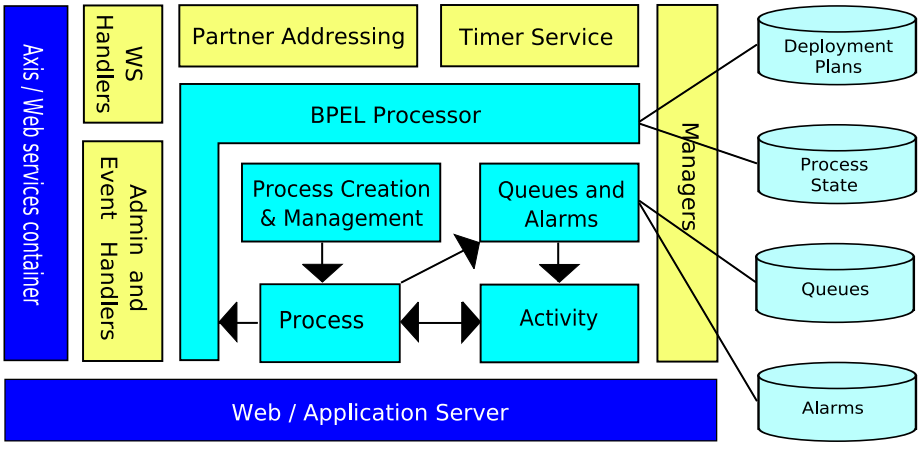


Figure 2.5: ActiveBPEL engine architecture

Figure 2.5 gives an overview of the ActiveBPEL architecture. The black boxes contain the components provided by the Apache Web server and the Axis Web services

containers. The dark grey boxes represent core components of the ActiveBPEL engine that are responsible for instantiating and executing a BPEL process. The light grey boxes represent components that are configurable and can be exchanged. Finally, the white boxes represent components that maintain persistent information about deployment plans, state of BPEL processes, queues and alarms.

Figure 2.6 shows what happens in the ActiveBPEL engine upon receiving a Web service request. The ActiveBPEL engine dispatches incoming messages to the correct process instance. A `receive` activity contains correlation data if a BPEL process is already active and waits for a `reply` activity after calling an `invoke` activity on an external Web service. In this case, the engine tries to find the correct instance that matches the correlation data. If there is no correlation data and the request matches a start activity, a new BPEL process instance is created.

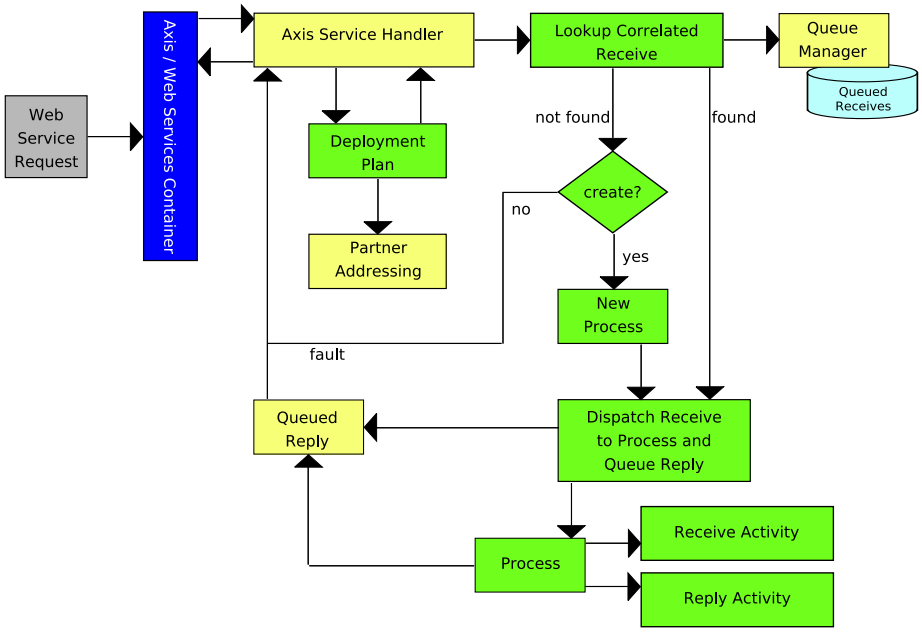


Figure 2.6: ActiveBPEL request dispatch flowchart

The receive queue contains the currently executing `receive` activities across all BPEL process instances. A `receive` activity is said to be executing when it has been queued by its parent activity (for example, a `scope`, `flow`, or `sequence`) but has not yet received the message that it is waiting for. The receive queue also contains messages that did not correlate to a waiting `receive` activity and did not correspond to a start activity. An unmatched receipt of data like this is possible given the asynchronous nature of some Web services. The engine will accept these unmatched messages provided that they contain correlation data. These messages are queued until a timeout period passes.

If a BPEL process queues an activity like a `receive`, then this activity will stay in the queue until the data arrives or the process terminates. A BPEL process terminates execution upon sending a final `reply` activity to the Web service consumer.

2.2.1.9 Partner Deployment Descriptor (PDD)

The ActiveBPEL engine defines `partnerLink` elements in a Partner Deployment Descriptor (PDD) file. Within the PDD file, the WS-Addressing [39] elements `EndpointReference` , `Address` and `ServiceName` are used to describe the endpoint of a WSDL service. A sample PDD file is shown in Listing 2.15.

Listing 2.15: Partner Deployment Descriptor (PDD) for ActiveBPEL engine

```
1  <process name="bpelns:bpel-process" location="bpel/bpel-
       process.bpel" ...>
2    <partnerLinks>
3      <partnerLink name="client">
4        <myRole allowedRoles="" binding="RPC" service="Client"
             />
5      </partnerLink>
6      <partnerLink name="WebService1">
7        <partnerRole endpointReference="static">
8          <wsa:EndpointReference xmlns:ws="urn:ws1">
9            <wsa:Address>ws1:anyURI</wsa:Address>
10             <wsa:ServiceName PortName="WPort">ws1:WebService</
                   wsa:ServiceName>
11           </wsa:EndpointReference>
12         </partnerRole>
13     </partnerLink>
14     <partnerLink name="WebService2">
15       <partnerRole endpointReference="static" invokeHandler=
             "process">
16         <wsa:EndpointReference xmlns:ws="urn:ws2">
17           <wsa:Address>ws2:anyURI</wsa:Address>
18             <wsa:ServiceName PortName="WPort">ws2:WebService</
                   wsa:ServiceName>
19         </wsa:EndpointReference>
20       </partnerRole>
21     </partnerLink>
22     ...
23   </partnerLinks>
24 </process>
```

The `partnerLink` that is specified in lines 3-5 is responsible for receiving a request from a service consumer, which initiates the creation of a new BPEL process within the ActiveBPEL engine (`myRole`). Lines 6-13 show the definition of a standard `partnerLink` that has an external WSDL endpoint whereas lines 14-21 define a `partnerLink` that refers to a locally deployed, on the same ActiveBPEL engine, service. This is indicated with the `invokeHandler="process"` statement.

2.2.1.10 BPEL Process Archive (BPR)

The BPEL Process Archive (BPR) is the packaging format that ActiveBPEL uses for deploying BPEL compositions to the engine. The structure is displayed in Figure 2.7.

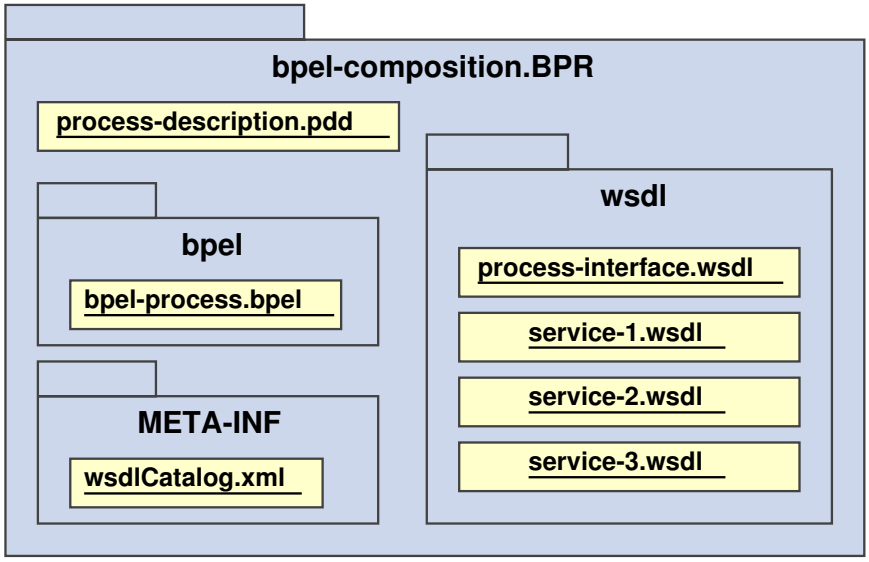


Figure 2.7: Contents of the BPEL Process Archive (BPR)

The description of the BPEL process is provided in the `bpel/bpel-process.bpel` entry. All WSDL interfaces that a BPEL service composition uses are contained in the `wsdl` folder. This folder also contains the WSDL interface that the BPEL service composition exposes to consumers, e.g. `process-interface.wsdl`. The ActiveBPEL engine uses a WSDL catalog to find WSDL files within a BPR, `META-INF/wsdlCatalog.xml`. The catalog references each of the wsdl interfaces in the wsdl folder. The PDD file, `process-description.pdd`, informs the ActiveBPEL engine about the BPEL process. It also defines required partner links and WSDL interfaces.

2.2.2 OWL-S

As part of the DARPA Agent Markup Language program (DAML) the Semantic Web community defines OWL-S (formerly DAML-S) [9]. OWL-S is based on the Web Ontology Language (OWL) [16] that defines a semantic markup language for publishing and sharing ontologies on the Web. OWL-S presents a Web service ontology with three interrelated sub-ontologies, known as the service profile, service model and service grounding (see Figure 2.8).

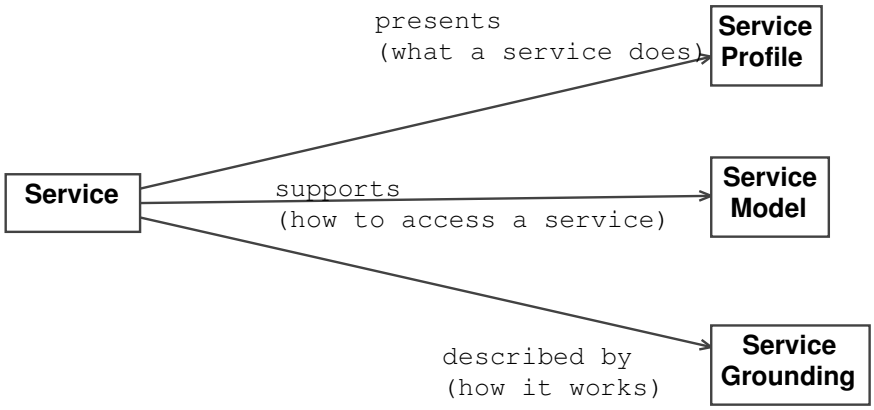


Figure 2.8: OWL-S service ontology

A service profile is used to advertise what a service does. The profile specifies the inputs, outputs, preconditions and effects (IOPE) of a service as well as additional information such as the service provider, quality of service and constraints [38]. The profile is only used to advertise a service in a registry in a machine-readable way. Once a service is selected the profile becomes useless and can be discarded.

A service model is used to describe how a service works viewing the service as a process. The process model is independent of any execution logic. OWL-S differentiates between atomic processes, simple processes and composite processes [38]. Atomic processes can be directly invoked and executed in a single interaction. The atomic process for the hotel service that was defined in Listing 2.7 for WSDL, is shown in Listing 2.16. A complex type is defined in the `owl:Class` element (lines 1-4) and all parts of the complex type are added separately with `owl:DataTypeProperty` elements (lines 5-12). The atomic process defines input and output parameters and their type.

Listing 2.16: Atomic process in OWL-S for hotel service

```
1  <owl:Class rdf:ID="BookingInformation">
2    <rdfs:label>BookingType</rdfs:label>
3    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/
         owl#Thing"/>
4  </owl:Class>
5  <owl:DatatypeProperty rdf:ID="name">
6    <rdfs:domain rdf:resource="#BookingInformation"/>
7    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
         string"/>
8  </owl:DatatypeProperty>
9  <owl:DatatypeProperty rdf:ID="dates">
10   <rdfs:domain rdf:resource="#BookingInformation"/>
11   <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
         string"/>
12 </owl:DatatypeProperty>
13 <process:AtomicProcess rdf:ID="HotelService">
14   <process:hasInput>
```

```
15      <process:Input ref:ID="In-bookingData">
16        <process:parameterType rdf:resource="#
              BookingInformation">
17      </process:Input>
18    </process:hasInput>
19    <process:hasOutput>
20      <process:Output ref:ID="Out-reservationNo">
21        <process:parameterType rdf:resource="&xsd;#string">
22      </process:Output>
23    </process:hasOutput>
24    <process:hasOutput>
25      <process:Output ref:ID="Out-successful">
26        <process:parameterType rdf:resource="&xsd;#boolean">
27      </process:Output>
28    </process:hasOutput>
29  </process:AtomicProcess>
```

Composite processes are composed of atomic or other composite processes. The composition can be defined by control constructs (`if-then-else` , `sequence` , `split` and `join`) [38]. Listing 2.17 illustrates a simple booking service that depends on two atomic processes. The depending processes are called sequentially.

Listing 2.17: Composite service in OWL-S

```
1  <process:CompositeProcess rdf:ID="Booking_Process">
2    <process:composedOf>
3      <process:Sequence>
4        <process:components rdf:parseType="Collection">
5          <process:AtomicProcess rdf:about="#bookFlight"/>
6          <process:AtomicProcess rdf:about="#bookHotel"/>
7        </process:components>
8      </process:Sequence>
9    </process:composedOf>
10 </process:CompositeProcess>
```

A simple process is an abstract process and cannot be invoked. It is used to provide an alternative view of an atomic process or an abstraction of a composite process. OWL-S was designed to be agnostic with respect to a process model formalism in order to remain compatible with emerging standards for process modelling like BPEL.

A service grounding specifies how a service can be accessed. Both the service profile and service model are abstract representations and the service grounding is responsible for mapping the abstract inputs and outputs of processes to concrete WSDL messages. Each atomic process must be provided with a service grounding. There can be more than one grounding for a service. The advantages of OWL-S are the rich semantics that enable automatic discovery, selection and composition. The WSDL grounding for the hotel service in Listing 2.16 is defined in Listing 2.18. Line 2 refers to the atomic process that this grounding represents. The `portType` and `operation` are defined (lines 3-12), followed by mapping input and output data to WSDL `message` elements (lines 13-38).

Listing 2.18: WSDL grounding in OWL-S for hotel service

```
<grounding:WsdlAtomicProcessGrounding rdf:ID="
    HotelServiceGrounding">
  <grounding:owlsProcess rdf:resource="#HotelService">
  <grounding:wsdlOperation>
    <grounding:WsdlOperationRef>
      <grounding:portType>
        <xsd:uriReference rdf:value="urn:hotelservice.wsdl#
            HotelPortType"/>
      </grounding:portType>
      <grounding:operation>
        <xsd:uriReference rdf:value="urn:hotelservice.wsdl#
            bookHotel"/>
      </grounding:operation>
    </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>
  <grounding:wsdlInputMessage rdf:resource="urn:hotelservice.
      wsdl#hotelRequest"/>
  <grounding:wsdlInput>
    <grounding:wsdlInputMessageMap>
      <grounding:owlsParameter rdf:resource="#In-bookingData"
          >
      <grounding:wsdlMessagePart>
        <xsd:uriReference rdf:value="urn:hotelservice.wsdl#
            bookingData">
      </grounding:wsdlMessagePart>
    </grounding:wsdlInputMessageMap>
  </grounding:wsdlInput>
  <grounding:wsdlOutputMessage rdf:resource="urn:hotelservice
      .wsdl#hotelResponse"/>
  <grounding:wsdlOutput>
    <grounding:wsdlOutputMessageMap>
      <grounding:owlsParameter rdf:resource="#Out-
          reservationNo">
      <grounding:wsdlMessagePart>
        <xsd:uriReference rdf:value="urn:hotelservice.wsdl#
            reservationNo">
      </grounding:wsdlMessagePart>
    </grounding:wsdlOutputMessageMap>
  </grounding:wsdlOutput>
  <grounding:wsdlOutput>
    <grounding:wsdlOutputMessageMap>
      <grounding:owlsParameter rdf:resource="#Out-successful"
          >
      <grounding:wsdlMessagePart>
        <xsd:uriReference rdf:value="urn:hotelservice.wsdl#
            successful">
      </grounding:wsdlMessagePart>
    </grounding:wsdlOutputMessageMap>
  </grounding:wsdlOutput>
  <grounding:wsdlVersion rdf:resource="http://www.w3.org/TR
      /2001/NOTE-wsdl-20010315">
  <grounding:wsdlDocument> "urn:hotelservice.wsdll" </
      grounding:wsdlDocument>
</grounding:WsdlGrounding>
```

2.2.3 Comparison of BPEL and OWL-S

While the main focus of BPEL is to provide an executable service composition the goal of OWL-S reaches much further into the intelligent and autonomous process area [7, 17].

One main disadvantage of BPEL is the dependence on static endpoints, which may lead to complete failure of a BPEL process if a single service becomes unavailable. Insufficient failure handling for an environment such as a DBE is another weak point. However, a BPEL process is executable, meaning that complete and open-source execution environments based on BPEL concepts exist, and BPEL receives industry-wide support.

On the other hand, OWL-S provides a more dynamic and intelligent way for defining service compositions. Required services are discovered at run-time rather then specifying static endpoints. However, no complete execution environment based on OWL-S concepts exists and there is a lack of support from industry partners.

Both approaches have their strengths and weaknesses. However, industry-support and an available open-source execution environment are definitively advantages that must be considered within a DBE as SMEs might not adopt proprietary and non standardised solutions.

2.3 Related Work

There are a number of academic research and industry projects that tackle problems concerning the dynamic and unstable availability of services at execution time. This section introduces four different approaches and compares their functionality.

2.3.1 METEOR-S

The METEOR-S project [35] of the LSDIS Lab, University of Georgia, provides tools to enable automatic service discovery and selection at deployment time based on constraints [4], which then uses the BPWS4J [12] engine to execute BPEL workflows. Service compositions are specified in BPEL but the required Web services are described in service templates [4]. Service templates define a semantic description of a service [46] as well as constraints for service selection such as user's preferences for service partners, QoS requirements, and service dependencies. An enhanced Universal Description, Discovery and Integration (UDDI) registry [57, 42] is queried with the service templates for matching service descriptions [49]. Matching service descriptions are further analysed if they meet the specified constraints [43]. Finally the best matching service is selected and with late-binding an executable BPEL process generated at deploy-time.

METEOR-S introduces a proxy [56] to support process configuration at both deploy and run-time. Also, the proxy is used to handle data and protocol level heterogeneities between used services at execution time. Each virtual partner of a BPEL process is represented by a proxy that has access to a *Protocol Mediator*, a *Data Mediator* and a *Process Configuration Module*. A remote call on an operation is delegated to the correct proxy instance as Figure 2.9 illustrates.

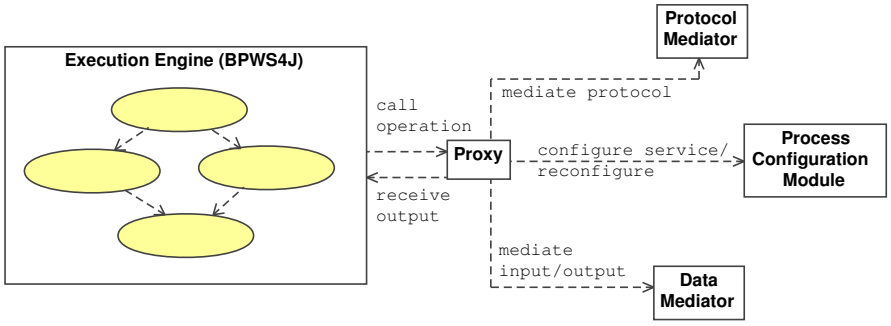


Figure 2.9: METEOR-S Proxy

A proxy uses the Process Configuration Module to retrieve details about the service and configure it. It contacts the Data Mediator to mediate the given data into the required data types. The Protocol Mediator is used to ensure that a service's interaction protocol is satisfied. If a service invocation fails the calling proxy initiates the reconfiguration of the BPEL process, replacing the service that failed. The replacement is conducted with two restrictions. As dependencies between services may exist, more than one service might have to be replaced. However, services that have already been executed cannot be replaced. All proxies of a BPEL process are therefore questioned for their state, which can be `executing`, `success` or `failure`. Waiting until no proxy has the state executing, the process manager halts all proxies until the reconfiguration process has finished. The reconfiguration process currently assumes that all services are replaceable.

2.3.2 OWL-S and Semantic Discovery Service

Another approach to overcome the static definition of services in a BPEL process and thereby enable dynamic discovery of required services at run-time has been proposed by McIlraith et al [38]. They introduce a Semantic Discovery Service (SDS) that serves as a dynamic proxy between the BPWS4J engine and service partners [36]. Figure 2.10 presents an overview of the invocation on an external service with the SDS.

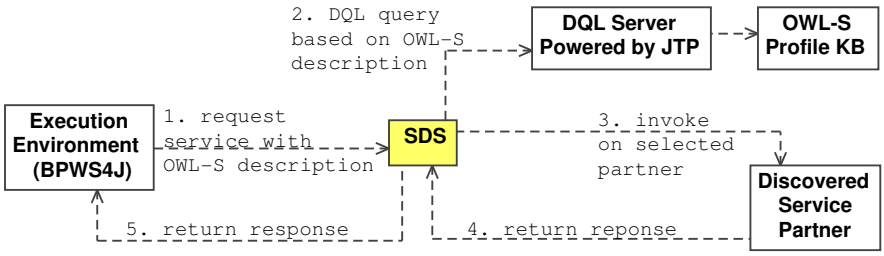


Figure 2.10: Invocation with SDS as proxy

Instead of invoking requests on statically bound partners the BPWS4J engine routes requests to the SDS. Service partners are described, advertised and discovered via a

OWL-S service profile. They adopt the DAML Query Language (DQL) [22] as the language and protocol to query a knowledge base of OWL-S service profiles. The SDS constructs a DQL query and sends it to the DQL server. The DQL server uses the Java Theorem Prover (JTP) [23] DAML + OIL reasoner to find service profiles in the knowledge base that match the query. Matching service profiles are returned to the SDS in answer bundles. The SDS chooses a matching service partner and invokes the partner's endpoint. Upon receiving a reply from the partner the SDS forwards the response to the BPWS4J engine. The SDS is stateless, i.e. has no knowledge about prior interaction, and agnostic to content of service descriptions and invocation messages.

2.3.3 Synthy

Synthy [3] decouples a Web service composition into a logical and a physical design stage. Figure 2.11 gives an overview of the two-staged approach that Synthy provides. The logical stage establishes a relationship based on semantic annotations between candidate services for a composition. Only service types are used at this stage to create abstract workflows that fulfil all functional requirements that the service specification specified. The set of abstract workflows is passed to the physical stage. The physical stage is responsible for creating one or more concrete BPEL workflows based on concrete service instances and QoS parameters relevant for this workflow that were specified in the service specification. Additionally, Synthy adds a fault-tolerant level to workflow execution. Two kinds of failures are identified, a binding may not exist for a given service type or a QoS requirement may not be satisfied. Multiple workflows can be created at the physical design stage and deployed with the runtime engine. These executable workflows can be ranked according to QoS information. In case of a runtime failure in a workflow the runtime engine can select the next highest ranked workflow to be executed.

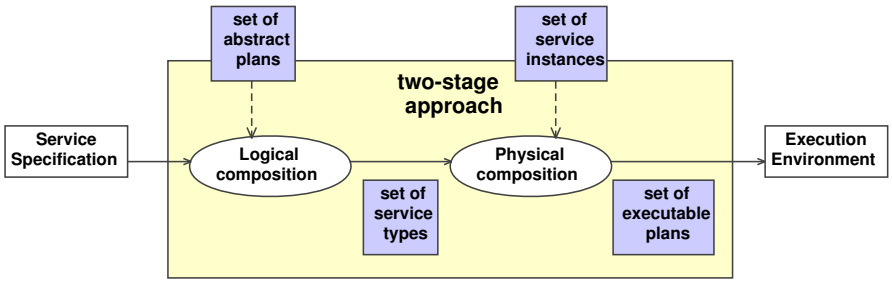


Figure 2.11: Synthy two-stage approach

2.3.4 Self-Serv

The Self-Serv framework [6] differs from the previously described approaches to the service composition problem. Self-Serv introduces a P2P-based orchestration model approach to the Web service composition problem. Self-Serv is based on two major concepts: composite services and service containers.

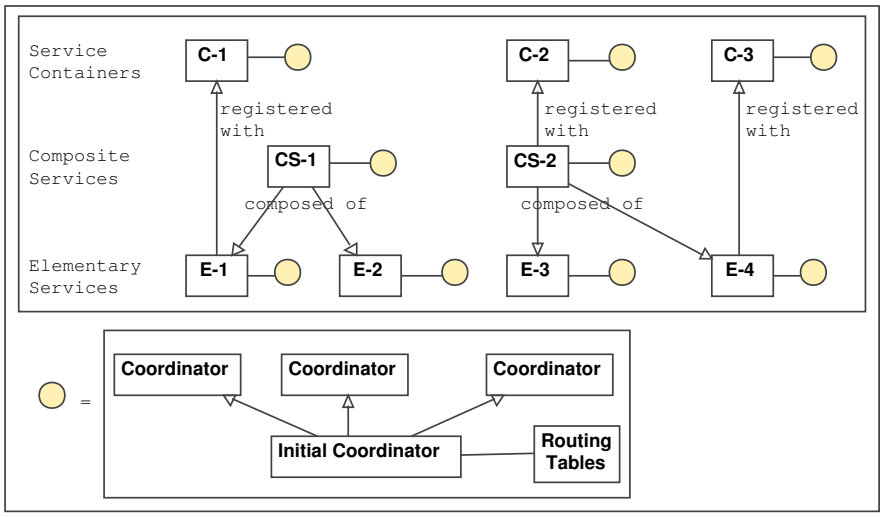


Figure 2.12: Self-Serv services

Figure 2.12 gives an overview of the service layer in Self-Serv. A composite service is composed of several other composite or elementary services. The business logic of a composite service is described in state charts. State charts were chosen within this approach because they possess all the formal semantics, are well-known and well-supported and can be easily transformed into other process modelling languages like BPEL. A service container is a service that aggregates several substitutable services. The aggregated services do not have to implement the same interface but must provide a mapping from the general service interface that a container supports and to their proprietary service interface. The container enables to measure the QoS for each offered service and perform load-balancing between the offered services. The service requests can be routed to the best matching, available and not overused service.

The P2P orchestration is based on state coordinators and routing tables. Self-Serv generates one state coordinator for each state in the state chart of a composite service. Each state coordinator is associated with a routing table that defines preconditions that have to be met for a state transition. The initial state coordinator initiates the composition and collects the results from the other states but is not responsible for state transitions. The composition is thereby self-orchestrated and not centralised.

2.3.5 Discussion

The three approaches METEOR-S, SDS and Self-Serv have in common that they enable dynamic service discovery at run-time. METEOR-S uses abstract BPEL processes and service templates, which contain semantic descriptions of required services, to enable dynamic service discovery at run-time. The description of services in service templates is a proprietary solution to METEOR-S. SDS intercepts all outgoing calls and discovers services via OWL-S service profiles. However, METEOR-S and SDS assume that services are replaceable but a service that cannot be discovered or replaced, will lead to the failure of a service composition in METEOR-S and SDS. Self-Serv can guarantee that services are replaceable with service containers. However, the concept

of service containers in Self-Serv has the negative impact that it can be a single point of failure.

Synthy differs from the other three approaches. Prior to deploying a BPEL process, service types are replaced with existing service instances. Multiple BPEL processes that offer the same functionality can be deployed with Synthy to provide redundancy on the one hand. On the other hand, deployed BPEL processes can be ranked, allowing Synthy to use the highest ranked process and replace a failed BPEL process with the next highest ranked one. This increases reliability of service compositions reliability and failure resilience.

	METEOR-S	OWL-S + SDS	Synthy	Self-Serv
service description	service templates	OWL-S service profiles	WSDL	WSDL
composition language	BPEL	BPEL	BPEL	state chart
execution environment	BPWS4J	BPWS4J	N/A	P2P
dynamic service discovery	yes	yes	no	yes
assumption that service instances replaceable	yes	yes	no	yes[a]
service composition reliability	no	no	yes	no
failure resilience	for single services	for single services	for service compositions	for single services

[a] ensured with service containers

Table 2.1: Summary of service composition approaches

From the preceding analysis of existing service composition approaches, see also Table 2.1, it can be seen that Synthy inspired the work of this deliverable most. A SME cannot make the assumption that services are always replaceable within the DBE, it is beyond their control. However, the METEOR-S and SDS approaches depend on this assumption. Although service containers in Self-Serv guarantee that services become replaceable, this approach relies on the collaboration of SMEs. With the Synthy approach, a SME can provide redundancy on the service providers side and thereby ensure better service composition reliability and failure resilience.

Our approach differs from Synthy in several ways: The coordination architecture enables BPEL service compositions to use WSDL services as well as DBE services. This enables us to discover instances of DBE services dynamically at run-time rather then relying only on static WSDL endpoints. The coordination architecture provides therefore failure resilience at service instance level as well as at service composition level.

2.4 Summary

This chapter presented an overview of the DBE, service compositions and related work in the area service reliability within BPEL service compositions.

Section 2.1 introduced the architecture of the DBE and its main components. As the work of this deliverable extends the deployment and consumption process of DBE services to support coordination of service compositions, these processes were described in detail in Section 2.1.3 and Section 2.1.4, respectively. A DBE service is deployed on an application server called the Servent [20]. An important feature of DBE services is the registration of a Proxy in a distributed lookup service (FADA) that service consumers can discover and download using an unique identifier, called SMID. The Proxy contains all information that a service consumer requires to invoke on a DBE service, such as endpoint information and a GUI. Essentially, the Proxy promotes loose coupling between clients and the protocol used to access a service.

The term service composition and a detailed description of the two main service composition languages, BPEL and OWL-S, is given in Section 2.2. The service composition problem can be described by two distinct phases, service composition creation and execution. Service composition creation includes decisions on the usage of different service description and composition languages whereas execution concentrates on service discovery, transaction mechanisms and failure handling. BPEL focuses on the execution aspects of a service composition whereas OWL-S focuses on describing services semantically to enable automatic service compositions.

Section 2.3 detailed four related works in the area of service composition. METEOR-S uses an abstract BPEL process and service templates to discover services at deploy- and run-time. BPEL process reconfiguration upon a service failure is also possible, under the assumption that services are replaceable (Section 2.3.1).

Section 2.3.2 presents a combination of describing services with OWL-S service profiles and BPEL. A SDS acts as a proxy and intercepts any calls for services from the BPEL engine. The OWL-S profiles are interpreted and available services discovered at run-time. This enables dynamic service discovery at run-time.

Another approach is Synthy, which partitions the service composition process into two steps, creating first an abstract BPEL process that uses service types. Available service instances are then used to create multiple BPEL processes that are deployed to a BPEL workflow engine. Synthy ranks the deployed processes and uses the highest ranked one. Upon a service failure, Synthy fails over to the next highest ranked BPEL process. Unlike the other approaches, Synthy enables service composition reliability and failure resilience at service composition level (Section 2.3.3).

Self-Serv uses state charts to describe service compositions and executes service compositions in a P2P fashion. One of the main ideas of Self-Serv are service containers. A service container aggregates replaceable services and measure their QoS. This enables a service container to delegate calls to the highest ranked service and perform load-balancing between services (Section 2.3.4).

Chapter 3

Design of the Coordination Architecture

In this chapter we discuss the main design decisions taken during the development of the coordination architecture. To tackle common issues of service compositions, we have identified a set of requirements for our architecture that are described in the first section of this chapter. This is followed by a description of user cases for our architecture and then the main body of our architecture. Section 3.3.1 gives an overview of the architecture and individual components. The deployment and execution of coordinated BPEL service compositions will be presented in Section 3.4.1 and Section 3.4.2 respectively. The advanced failure handling that the architecture provides is described in Section 3.4.3. Section 3.4.4 introduces the transition between different service types. Finally, the last section gives a summary of this chapter.

3.1 Design Requirements

As introduced in Section 2.2, we have identified a set of requirements based on the analysis of the DBE and existing service composition solutions that aim to provide a guide. The following list refines the requirements in more detail and aims at providing a guide for the design of the coordination architecture.

1. Open-source and standards-based:
   The DBE is released as an open-source project in order to allow providers of Web services to integrate their services adequately and to allow the verification of code in terms of security. In order to facilitate the integration of Web services, the DBE needs to be based on open standards. This allows providers to offer Web services with the same or similar functionality while guaranteeing that these services are interchangeable. This led to BPEL being chosen as the service composition language within the work of this deliverable. BPEL is currently in the process of being standardised and is already widely accepted for defining executable business processes in industry. Another advantage of BPEL over other service composition approaches is that there are open-source BPEL workflow engines available, such as the ActiveBPEL engine.

2. Ranking of available services:
   Service providers can offer many functionally equivalent service compositions to

service consumers that only differ in their QoS information. To allow consumers to use the service with the highest QoS, the coordination architecture should exploit this QoS information to create a ranking of available services. Further on, the ranking information for the available services should be retrieved and updated automatically by the architecture in order to ensure that the architecture always uses the highest ranked service and thereby ensure high QoS to service consumers. In order to address this requirement, we will introduce a ranking service that is integrated within the workflow engine.

3. Service reliability and failure resilience:
   Failures are not acceptable for service compositions within the DBE so one of the main requirements is that the architecture offers adequate failure handling. As services are likely to fail within the DBE, leading to partial or complete failure of a service composition, the architecture should switch automatically to alternative services upon a service composition failure to increase service reliability and failure resilience. This requires our architecture to provide fail-over for service instances as well as fail-over for service compositions.

4. Ease of use for service consumers:
   Service consumers want faultless services that are easy to use. Any changes that this architecture introduces should therefore remain hidden from service consumers. Likewise, the architecture should keep a service consumer unaware of most service failures and problems to increase the ease of use for service consumers.

3.2 User Cases

The following two use case diagrams introduce the functional model of the coordination architecture from two perspectives: The service provider and the service consumer perspective.

3.2.1 Service Provider Perspective

A service provider has three tasks regarding a Web service composition. The tasks and their subtasks are illustrated in Figure 3.1.

- Creation of a Service Composition
  A service provider must first decide on the functionality that a new service composition will offer to consumers. The process of creating a service composition can be split up into several subtasks: such the search for adequate services, the choice of services to use and the order in which services must be used.
  A service composition uses any number of services within a BPEL process. In order to include these services into a service composition, the service provider searches for already available services within the DBE. Available services must then be compared by the service provider to select those that best match the purpose of the new service composition. If several services match the requirements, the service provider uses other important non-functional criteria for selecting a service, such as the quality of a service.
  The service provider is assisted during these tasks by the CLE. The CLE offers

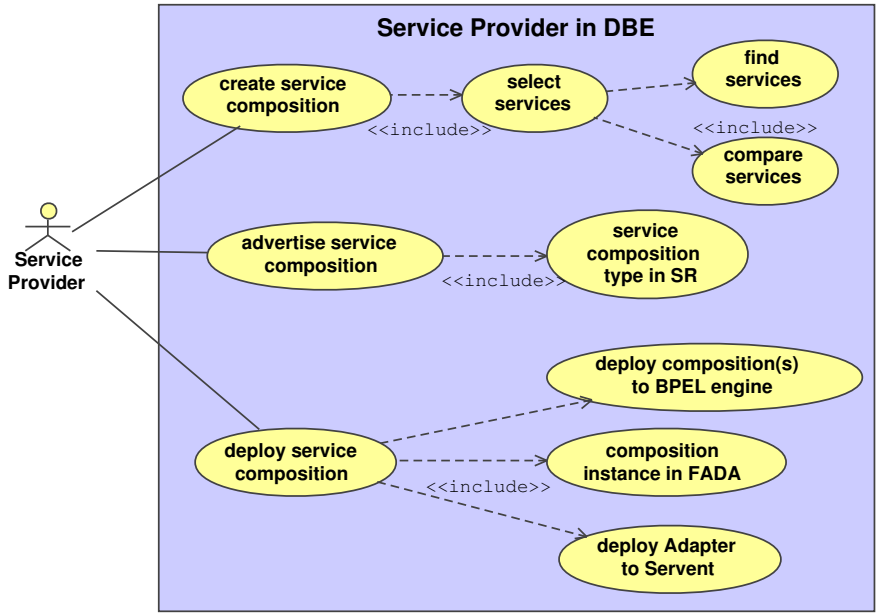


Figure 3.1: Use Case Diagram of Service Provider

a set of tools and wizards that can be used to discover, compare and select available services for inclusion in the service composition. This tool already exists in the DBE studio and was created outside the scope of this deliverable.

- Advertisement of a Service Composition
  The service provider advertises the newly created service composition to the Semantic Registry. This advertisement includes only service type information and does not specify any service instance information. The Semantic Registry returns a unique SMID for the advertised service to the service provider.
  The advertisement process is already defined within the DBE and was created outside the scope of this deliverable.

- Deployment of a Service Composition
  In order to provide an instance of the new service composition to consumers, the service provider deploys the service composition to a Servent instance using the ET. This involves three tasks: The Servent deploys one or more BPEL service compositions to a BPEL workflow engine. Then the Proxy object for this service composition instance is registered with FADA using the SMID as a unique identifier. Finally, the Servent deploys the Adapter for the service composition locally on the Servent.
  The deployment process for DBE service is modified within the work of this deliverable to support the deployment of service compositions as well. The deployment will be defined in detail within this chapter.

3.2.2 Service Consumer Perspective

A service consumer perspective exposes two main operations for service consumers, depicted in Figure 3.2.

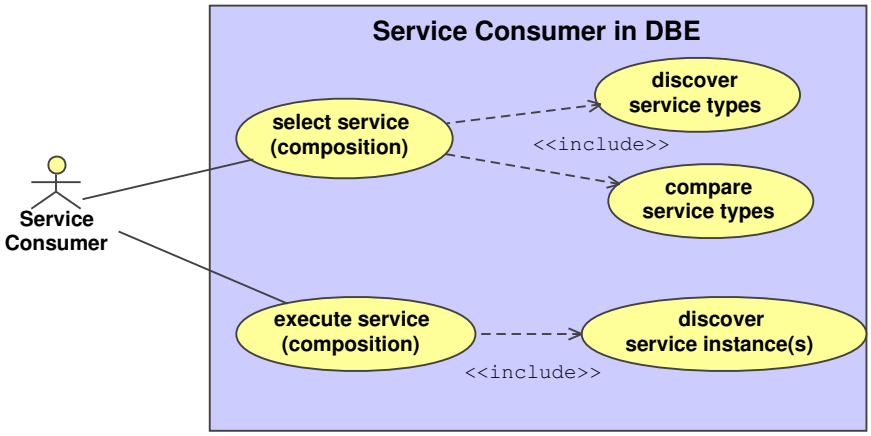


Figure 3.2: Use Case Diagram of Service Consumer

- Selection of a Service (Composition)
  The service consumer requires a service to complete a task such as a booking service to organise a journey. After specifying certain criterions about the required service, the service consumer contacts the DBE to discover suitable services. If several services within DBE are suitable, the service consumer compares the available services to find a service is for example the highest-ranking or the cheapest one. Upon discovering and comparing available services, a service consumer selects one of the available service to complete the task and retrieves its SMID.
  The selection process is already defined within the DBE and was created outside the scope of this deliverable.

- Execution of a Service (Composition)
  Once the service consumer has chosen a service, it uses the SMID to discover an instance of the selected service on FADA. If the chosen service represents a service composition, the service composition will automatically retrieve service instances for all used services within its composition from FADA. However, the service consumer remains unaware of this process unless a failure occurs because a service instance is unavailable.
  We will define the execution of composed services within this chapter in detail as the process differs from the execution of DBE services.

3.2.3 Summary

Within this section we defined the functional model of the coordination architecture. As our work extends work from other partners within the DBE, we identified the operations that need modification to support the coordination architecture, such as the deployment and execution process of service compositions.

3.3 Coordination Architecture

The following sections provide an overview of the architecture for coordinating service compositions in the DBE and its components. First, we present overall coordination architecture. Subsequent sections deal with the deployment and execution details of coordinated service compositions and the design of individual components.

3.3.1 Architecture Overview

In this section, we give an overview of the architecture for coordinating BPEL service composition in the DBE. As the work of this deliverable is part of a larger EU project, certain components that have been build for this project can be reused or extended. Figure 3.3 outlines the architecture components for coordinating BPEL compositions in the DBE, where white components depict reused ones, grey components extended and black components added ones. All relevant components will be described in detail in subsequent sections.

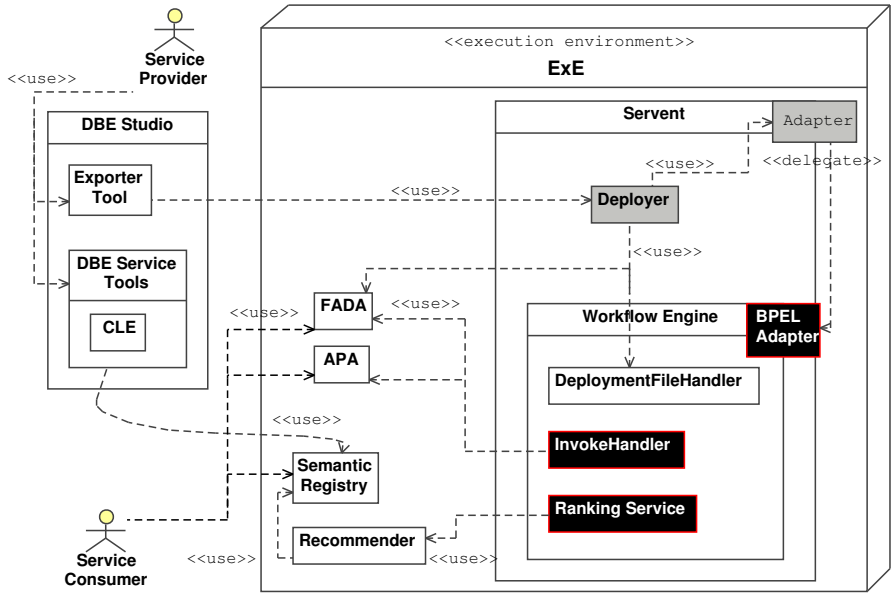


Figure 3.3: Architecture overview

As described in Section 2.1, the DBE Studio [19] is a development environment for the DBE that provides among other things a collection of DBE service tools and an ET. A service provider can use the DBE Studio to analyse business services, define business and service models, develop DBE services and finally deploy and publish DBE services to the ExE. Part of the DBE Service Tools is the CLE, which offers amongst other tools an editor to create BPEL service compositions.

Once a service provider has created a DBE service, the DBE service has to be registered with the Semantic Registry. The ET is then used to export a DBE service to the Servent [20]. The ET packages a DBE service in conjunction with a deployment descriptor into a *DBE archive* (DAR). The DAR is handed over to the Deployer of the

Servent that handles the deployment. The Deployer deploys a DBE service locally on a Servent. If a DAR file contains one or more BPEL service compositions each of them is deployed to a workflow engine via the `DeploymentFileHandler` interface. The Deployer then registers the DBE service with the Adapter and advertises it to FADA.

A service consumer uses the Semantic Registry to search for available DBE service types. Instances of a selected DBE service can then be discovered via FADA. Invocations for a BPEL service compositions are intercepted in the Adapter of a Servent and delegated to the BPEL Adapter. The BPELAdapter delegates the call to the desired BPEL composition after making any necessary transformations. A BPEL service composition might use several DBE services within its composition. The Invoke Handler is used by compositions to make invocations on any external DBE services, retrieving required DBE service instances from FADA and making the invocation via the APA. The Ranking Service can be contacted by a coordinating BPEL service composition to retrieve the highest ranked service out of a set of deployed BPEL service compositions.

3.4 Deployment and Execution

3.4.1 Deploying Coordinated Service Compositions

In order to deploy DBE service that represent service compositions, we had to extend the generic deployment process that was described in Section 3.4.1 as several BPEL service compositions must be deployed to a BPEL workflow engine in a specific order and the adapter implementation must be registered with a different key.

As depicted in Figure 3.4, the ET commissions the Deployer of a Servent to deploy a DBE service. The deployment descriptor specifies explicitly that the DAR file of this DBE service contains one or more BPEL service compositions. As one BPEL service composition can depend on the successful deployment of another, the deployment descriptor also specifies in which order the BPEL service compositions must be deployed. Each BPEL composition is deployed to the workflow engine via the `DeploymentFileHandler` interface. Although this interface is specific to the ActiveBPEL engine, a similar interface should be available in any BPEL workflow engine to deploy a BPEL service composition. Thereby the deployment process remains independent of a specific version or type of BPEL workflow engine.

Upon successful deployment, the Deployer registers the DBE service for the coordinated BPEL service composition with the Adapter. The Adapter identifies DBE services with a unique key. The key for DBE services that represent a BPEL composition is enhanced with a BPEL specific extension. This extension informs the Adapter if a call for a DBE service must be delegated to the BPELAdapter.

The Deployer then advertises a Proxy for the DBE service in FADA so that service consumers can find the DBE service. DBE services that represent a BPEL service composition do not differ in this behaviour from generic DBE services.

3.4.2 Executing Coordinated Service Compositions

In this section we presents the coordination of BPEL compositions at execution time in detail, followed by a detailed description of the used components. Section 2.1.4 described the execution process for generic DBE services. However, the execution of coordinated BPEL service compositions differs from this process as incoming requests must be delegated to a BPEL service composition. Similar to the execution of a simple

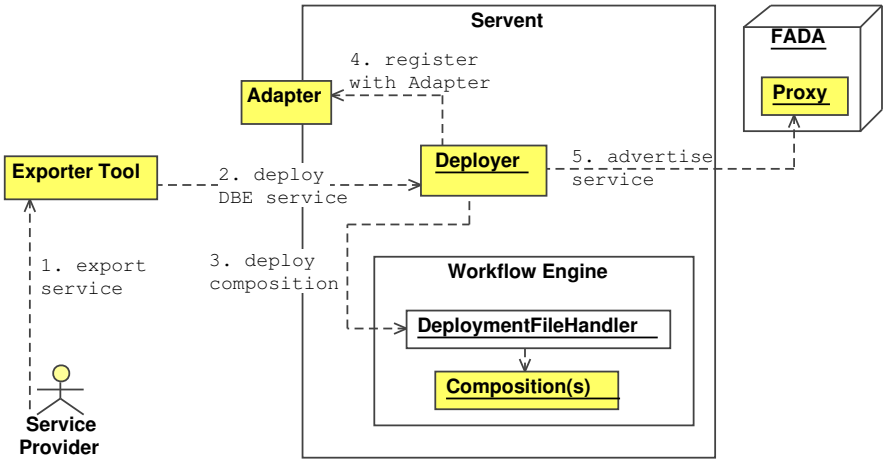


Figure 3.4: Deployment and advertisement of BPEL compositions

DBE service, a BPEL service composition is called by a service consumer via its Proxy. The Servent routes the call to the BPELAdapter that transforms the request into a SOAP call and then delegates it to the called BPEL service composition instance. The BPEL service composition uses the Ranking Service to retrieve the highest ranked service composition and executes the service consumer's request using this service. Figure 3.5 gives an overview of the components that participate in the execution of a BPEL service composition with the coordination architecture.

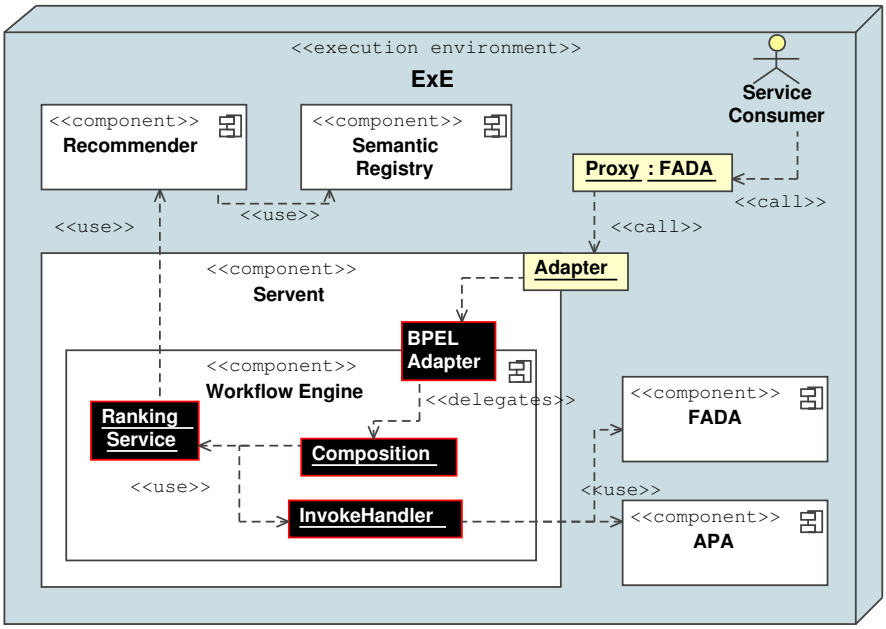


Figure 3.5: Execution of a Service Composition

3.4.2.1 BPEL Adapter

The Servent currently uses object serialisation as the default protocol for DBE services whereas BPEL service compositions use the SOAP protocol. The usage of different protocols are masked for service consumers in the Proxy and APA [15] and the call is delegated to the Adapter in the Servent as depicted in Figure 3.6. The Adapter has access to all DBE services that are deployed within a Servent instance and identifies them with a unique key.

In order to differentiate BPEL service compositions from generic DBE service, we extended the key for DBE services that represent a BPEL service composition with a BPEL specific extension. Upon retrieving a DBE service with such a key, the Adapter delegates the incoming call to the BPELAdapter. Calls for simple DBE services are still delegated to their endpoint within the Servent. The BPELAdapter automatically converts object serialisation calls into SOAP calls and then forwards the client requests to the BPEL workflow engine.

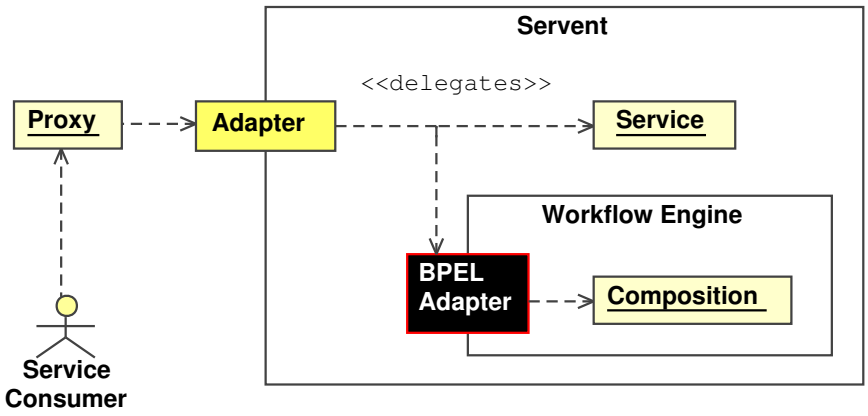


Figure 3.6: BPEL-Adapter for Servent

3.4.2.2 Coordinating BPEL Service Compositions

Within our architecture, a set of BPEL service compositions that provide equivalent functionality may be deployed locally on a BPEL workflow engine. These service compositions are coordinated by one BPEL service composition that represents a service type. As shown in Figure 3.7, the coordinator receives incoming requests from a service consumer and starts to coordinate the execution. In order to determine the highest-ranking BPEL service composition out of the deployed set of BPEL service compositions, the coordinator first contacts the Ranking Service. The Ranking Service contacts the Recommender that maintains a history of the performance of DBE services, including coordinated BPEL service compositions, and is returned the highest-ranking service. The Ranking Service returns this information to the coordinator. Depending on the service type of the chosen BPEL compositions, the coordinator might have to transform some or all types of the consumer request into the required types for the selected BPEL service composition. This can either involve simple type conversions within the BPEL process or calling other Web services to obtain the required

values and types. The coordinating BPEL service composition then delegates the request to the selected BPEL service composition.

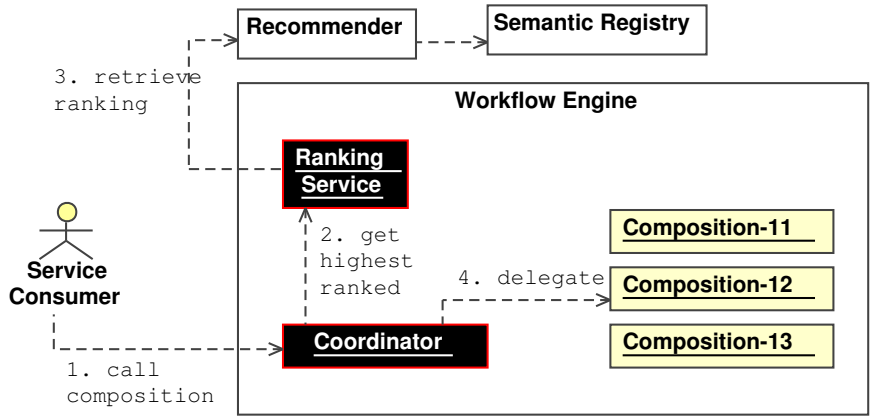


Figure 3.7: Coordination of BPEL compositions

3.4.2.3 Ranking Service

We introduce the Ranking Service within our architecture to retrieve the currently highest ranked service out of a set of deployed BPEL service compositions. The Ranking Service is a Web service that is installed as a core service on any BPEL workflow engine within the DBE. As shown in Figure 3.7, the coordinator uses the Ranking Service to retrieve the currently highest ranked service out of a set of deployed BPEL service compositions. To achieve this task the Ranking Service is given a unique identifier of each BPEL service composition and contacts the Recommender with these identifiers. The Recommender, after contacting the Semantic Registry, returns the highest ranked BPEL service composition out of the provided set to the Ranking Service.

Upon a failure of a BPEL service composition, the coordinator contacts the Ranking Service again and passes in information about the failed service. The Ranking Service delegates the failure information to Semantic Registry via the Recommender and, upon receiving the next highest ranked BPEL service composition, returns this to the coordinator.

We designed the Ranking Service to be very lightweight so that it does not leave a heavy footprint on SMEs with low ICT access. But although contacting the Ranking Service is not time consuming itself, retrieving the current ranking via the Recommender from the Semantic Registry depends on many other factors within the DBE such as network speed and connectivity. The discovery of Recommender and Semantic Registry instances and potential network delays are outside the scope of this deliverable.

3.4.2.4 Customising the Invoke Handler

A BPEL service composition starts to execute when the `receive` activity is invoked. It can then use a number of other services, calling them with the `invoke` activity. The `reply` activity is used to return results, if there are any, to the service consumer

and finish execution. Within our architecture there are three possible service types that a BPEL service composition can call, as shown in Figure 3.8.

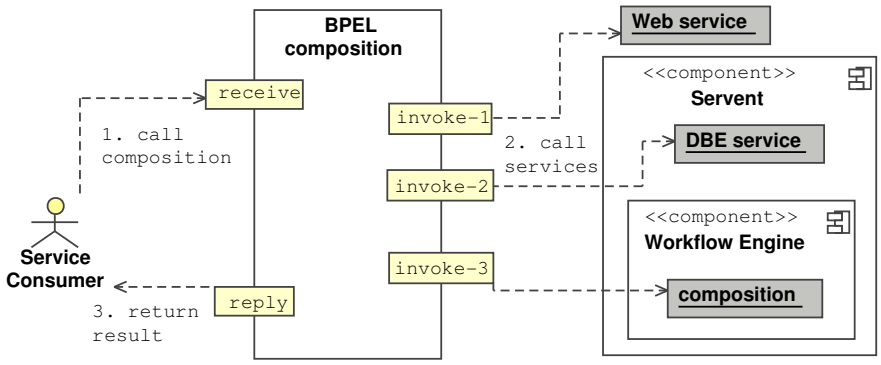


Figure 3.8: Invoke activity of BPEL compositions

Calling a Web service is the standard activity within a BPEL service composition and therefore supported in any BPEL workflow engine. However, DBE services, whether they are simple or composed services, are discovered via their SMID and used via a Proxy. Therefore, we required a customised InvokeHandler that implements the `invoke` activity within a BPEL workflow engine. The standard InvokeHandler is extended to call DBE services as well as WSDL services. As illustrated in Figure 3.9, the customised InvokeHandler uses a SMID to download the corresponding Proxy object for an available DBE service from the FADA at runtime. The invocation is then delegated to the APA. Essentially, the InvokeHandler uses a generic way to consume services in the DBE, similar to what was described in 2.1.4.

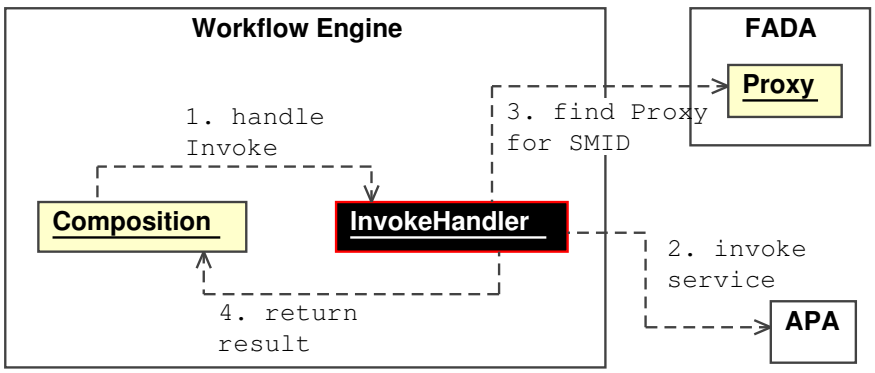


Figure 3.9: Customised Invoke Handler

3.4.3 Failure Handling Mechanisms

We identified failure handling as one of the main requirements for a coordination architecture as the expected failure rate of services in the DBE can be relatively high. There

are two possible failure models for a service composition within the DBE that must be considered within our architecture. First, a BPEL service composition depends on a DBE service that fails but can be replaced with another instance. We will refer to this mechanism as *service instance fail-over*. The second failure model occurs when the failure of one or more DBE services lead to a complete failure of a BPEL service composition. We will refer to this mechanism as *service composition fail-over*.

We introduce the service instance fail-over mechanism as a more fine-grained failure handling technique. This mechanism is depicted in Figure 3.10. This technique takes advantage of the special circumstance that several DBE service instances, representing instances of the same service type, can be registered in FADA with the same SMID. In this case, the InvokeHandler is returned a Proxy for all available service instances with the same SMID. If a used service instance fails at invocation time, the InvokeHandler catches the failure and attempts to fail over to one of the other service instances. Only if all available service instances fail, the complete BPEL service composition fails resulting in the second failure model.

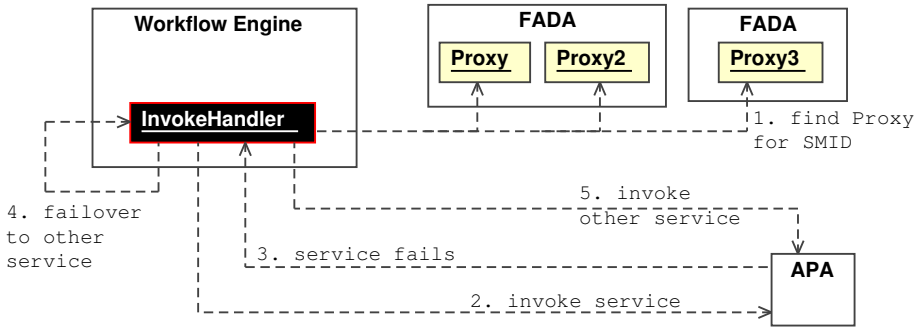


Figure 3.10: Fail-over to another service instance in the InvokeHandler

If a complete BPEL service composition fails, our architecture uses the service composition fail-over mechanism. We introduced the coordinator to catch and handle such a failure. As our architecture encourages service providers to supply more than one BPEL service composition available, the coordinator checks whether all coordinated BPEL service compositions have failed. In this case a failure message will be returned to a service consumer. Otherwise the coordinator contacts the Ranking Service again to retrieve the next highest ranked BPEL service composition. The coordinating BPEL service composition then continues executing the service consumer's request by delegating the request to the next highest ranked BPEL service composition. Upon successful execution the coordinating BPEL service composition returns the result to the service consumer. This failure model is shown in Figure 3.11.

The advantages of this twofold failure handling are that we can guarantee service consumers an increased service reliability and failure resilience, thereby increasing the ease of use for service consumers. On the other hand, service consumers might experience some delay until the architecture can detect a failure and use one of the two failure handling methods to continue executing a service consumer's request. If the first failure handling mechanism can be used successfully, the experienced delay for a service consumer should be minimal. However, the second failure model can cause a much higher delay as the used BPEL service composition might have to be switched

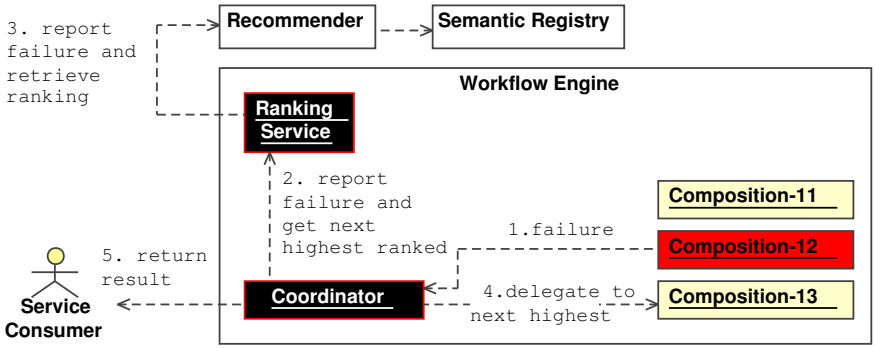


Figure 3.11: Advanced failure handling

several times by the coordinator.

3.4.4 Transitioning between Different Service Types

In this section we describe how the designed architecture enables service providers to encapsulate different service types for BPEL service compositions. An important constraint is that all required types must be either type-compatible or type-convertible, meaning that the given types can be transformed into the required ones. A service provider is responsible for defining the transition steps from one BPEL service composition to another in the coordinating BPEL service composition.

Different methods are possible for accomplishing a transition. The simplest transition simply requires the combination of available types via methods such as adding or subtracting their values. A very simple example for such a transition are two booking services where one differentiates between adults and children and the other does not. To transition from the first type to the second type the number of adults and children simply would have to be added. More complicated transitions might involve invoking other Web services from the coordinating BPEL service composition to retrieve the required types.

All transitions must be based on BPEL syntax to remain standard compliant. The BPEL specification defines that XPath expressions must be supported by any standard-compliant BPEL workflow engine. XPath expressions can be used to retrieve values from variables and add them, for example. Another way to support the transition from one type to another is the invocation of additional services in a BPEL process.

3.5 Design Summary

In this chapter we described the architecture for the coordination of service compositions in the DBE, its use cases and the overall architecture with a detailed view on the deployment and execution aspect of coordinated BPEL service compositions, advanced failure handling mechanisms and transitioning between different service types.

The coordination architecture has two main perspectives that we presented in Section 3.2. The service provider is concerned with creating, deploying and advertising service compositions. The goal of the coordination architecture is to enable service providers

to provide service compositions that are failure resilient and have a high reliability and availability. The service consumer is mainly interested in the ease of use for discovering and executing services that work faultless and provide good performance. From that point of view, the coordination architecture should be seamlessly integrated into the service provider's service composition without interrupting the service consumer's perception of the service composition behaviour.

Section 3.3.1 presented a high-level overview of the coordination architecture by providing the functional model describing the use cases of both service provider and service consumer. Subsequent sections dealt with the specifics of the deployment and execution aspects of coordinated service compositions and the design of each participating component. The coordination architecture basically consists of the BPELAdapter, the coordinating service composition, the Ranking Service, the customised InvokeHandler and advanced failure handling mechanisms. With the help of the BPEL-Adapter, a call on a generic DBE service is converted into a request for a BPEL service composition (Section 3.4.2.1). The coordinating service composition provides an entry point to a set of BPEL service compositions (Section 3.4.2.2). It uses the Ranking Service to retrieve the highest ranked BPEL service composition from the set of available ones. The coordinating BPEL service composition provides one of the two failure handling mechanisms. If a selected BPEL service composition fails, the coordinating BPEL service composition catches this failure and attempts to use the next highest ranked one. The Ranking Service obtains the ranking order for available BPEL service compositions and updates the ranking upon a service failure (Section 3.4.2.3). The customised Invoke Handler enables service providers to use DBE services as well as Web services within a BPEL service composition (Section 3.4.2.4). As DBE services are not identified via a static endpoint, the Invoke Handler can fail over to another service instance if more than one service instance is available for a specific service type. This is the other failure handling mechanism that the architecture uses to improve failure resilience of service compositions.

The next chapter presents the implementation of the architecture for coordinating BPEL service compositions in the DBE, using the ActiveBPEL engine for execution of BPEL service compositions.

Chapter 4

Implementation of the Coordination Architecture

In this chapter, we presents the implementation of the architecture for coordinating service compositions in the DBE. The implementation is based on the requirements that we set out in Section 3.1.

The source-code presented in this chapter does not contain any logging messages or error handling logic. As well, for the sake of brevity, constants have been replaced by their actual value.

The first section of this chapter details the development environment that was used to create the implementation. This is followed by a description of the components that have been introduced to the existing architecture and the modifications that have been made to existing components. The chapter concludes with a summary of what has been achieved.

The result of this implementation is part of the open-source project Swallow [21] that also contains the ServENT implementation [20].

4.1 Development Environment

The coordination architecture has been developed based on J2SE 1.4.2 (build 1.4.2_08), BPEL 1.1 [5] (see Section 2.2.1), AXIS 1.2.1 [27], WSDL 1.1 [8], the extended ServENT 0.2.0 [20] (see Section 2.1.3 and Section 2.1.4) and ActiveBPEL 2.0 [2] (see Section 2.2.1.8).

Figure 4.1 illustrates the development environment of this project. The source-code for this deliverable was maintained with CVS [1]. The Maven [24] and ANT [26] build systems were used to manage the whole project. Eclipse 3.1 [28] was used as the development environment.

4.2 Coordination Architecture

In order to support the coordination architecture, we had to introduce components to the existing architecture and to modify some existing ones. Figure 4.2 outlines new components in black, grey components modified ones and white ones could be reused. All relevant components will be described in detail in subsequent sections.

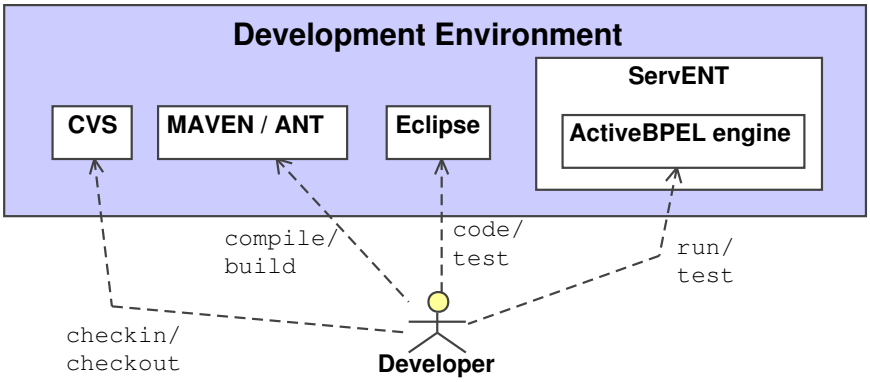


Figure 4.1: Development Environment

The `ClientHelper` class shrouds service consumers from FADA details. Previously, the `ClientHelper` class returned only one Proxy object to consumers. In order to support service instance fail-over, we extended the class to return all Proxy objects that match a SMID. The `DeployerImpl` class is responsible for deploying DBE services on the ServENT and for registering Proxy objects with FADA. This class is extended to deploy BPEL service compositions to the ActiveBPEL engine. The `DeployerImpl` uses the `ServiceConfiguration` class to read in the deployment descriptor of DBE services. In order to support service composition relevant entries in the deployment descriptor, the `ServiceConfiguration` had to be updated. In order to support DBE services as well as Web services, the standard `InvokeHandler` of the ActiveBPEL had to be customised. The `SSSHandler` class is responsible for receiving requests from consumers. It correlates incoming requests or rather their key with the corresponding services. We extended the `SSSHandler` to recognise a service composition key and to delegate the call correctly to the `BPELAdapter`. The `BPELAdapter` component was introduced to convert object serialisation calls that the ServENT uses to SOAP calls for the ActiveBPEL engine. Finally, the Ranking Service was introduced to support the ranking of the coordinated service compositions.

4.3 Deploying Coordinated Service Compositions

4.3.1 DBE Archive

Figure 4.3 gives an overview of the DBE Archive (DAR) that is used as the format to deploy a packaged DBE service to the ServENT.

A DAR contains a service implementation and a deployment descriptor as described in Section 4.3.2. We extended the deployment descriptor for the coordination architecture to specify all service information for coordinated BPEL service compositions. The `service-composition` folder is an extension to the DAR format and contains any BPR files and the WSDL interface of the coordinator in the `service-composition/wsdl` folder. The WSDL interface is used by the implementation of the BPELAdapter to create a SOAP call. The `CoordinatorServiceAdapter`

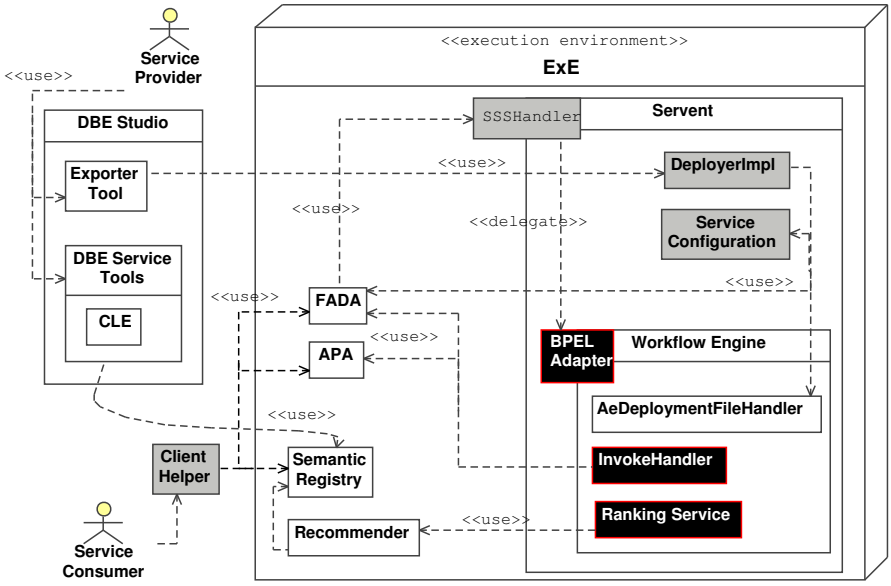


Figure 4.2: Architecture overview

class contains a Java representation of this WSDL interface. The adapter class is required within the Servent to create the `AdapterWrapper` object and the Proxy object that represent this DBE service but requires no implementation as all calls are delegated to the BPELAdapter.

4.3.2 Deployment Descriptor

The deployment descriptor of a DBE service is used to describe any service-related information. Listing 4.1 displays the contents of an enhanced deployment descriptor for the coordination architecture.

Listing 4.1: Contents of an enhanced deployment descriptor

```
<deploymentProperties>
    <serviceInfo>
        <serviceName>Coordinator</serviceName>
        <smid>coordinating-process-SMID</smid>
        <adapterClass>org.dbe.service.CoordinatorServiceAdapter
            </adapterClass>
        <complexTypes>
          <complexType>org.dbe.service.SomeComplexType</
              complexType>
        </complexTypes>
    </serviceInfo>
    <compositionInfo>
        <wsdlLocation>service-composition/wsdl/coordinator-
            interface.wsdl</wsdlLocation>
        <wsdlNamespace>urn:service.dbe.org</wsdlNamespace>
        <wsdlService>Coordinator</wsdlService>
```

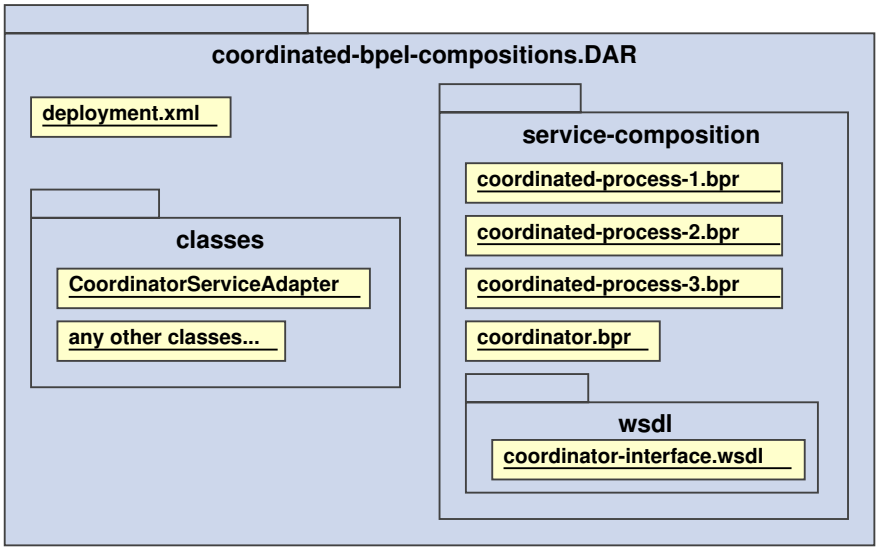


Figure 4.3: Contents of a DBE Archive (DAR)

```
14          <redirectTo>CoordinatorBPEL</redirectTo>
15          <bprServices>
16              <bpr>coordinated-process-1.bpr</bpr>
17              <bpr>coordinated-process-2.bpr</bpr>
18              <bpr>coordinated-process-3.bpr</bpr>
19              <bpr>coordinator.bpr</bpr>
20          </bprServices>
21      </compositionInfo>
22  </deploymentProperties>
```

The first part contains the standard information, name of a service, SMID with which the Proxy of a DBE service is registered in FADA and the adapter class for the Servent. A `complexTypes` element was added to create the Java class of a complex type in the customised Invoke Handler, which is required for an object serialisation call on a DBE service.

The second part describes the composition information:

- `wsdlLocation` : This entry describes the location of the WSDL interface for the coordinator. The interface is used by the BPELAdapter to create a SOAP call.
- `wsdlNamespace` and `wsdlService` : These two entries are used by the BPELAdapter to create the correct `org.apache.axis.client.Service` object from the WSDL interface.
- `redirectTo` : An incoming call to the Servent for a BPEL service composition has to be redirected to another endpoint. The Servent knows the general endpoint of ActiveBPEL services, a concatenation of the private URL of the Servent `http://hostname:2728/` with `active-bpel/services` . The

`redirectTo` part gives the last identifier, such as `Coordinator`, of the new URL. So the complete URL where the BPELAdapter has to delegate the invocation to is `http://hostname:2728/active-bpel/services/Coordinator`.

- `bprServices`: This entry gives the order in which the coordinated BPEL compositions and their coordinator must be deployed to the ActiveBPEL engine.

4.3.3 Deployer

The previous sections described the extended DAR file and deployment descriptor formats that are used during the deployment of a composed DBE service to a Servent instance. Figure 4.4 illustrates the invoked methods in a sequence diagram.

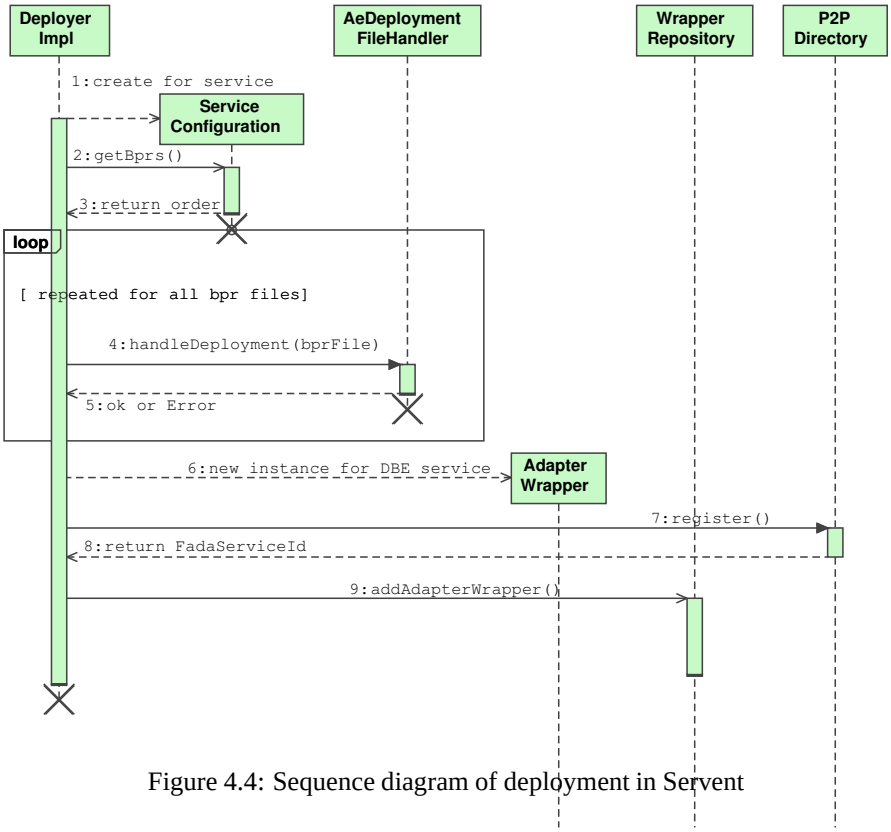


Figure 4.4: Sequence diagram of deployment in Servent

DBE services are kept in a specific `deploy` folder within a Servent. Upon startup or on receiving a new DAR file, the `DeployerImpl` class creates a `ServiceConfiguration` object for a DBE service that contains all information from the corresponding deployment descriptor. We had to extend the `ServiceConfiguration` object with relevant information for the coordination architecture, such as the order in which the BPR files must be deployed to the ActiveBPEL engine. This was necessary as the coordinator depends upon successful deployment of all coordinated BPEL service compositions and must therefore be deployed last. The `DeployerImpl` uses the `AeDeploymentFileHandler` class in the ActiveBPEL engine to deploy BPEL compositions in the BPR format.

Once all BPEL service compositions are deployed to the ActiveBPEL engine, the `DeployerImpl` creates an `AdapterWrapper` for the DBE service and registers the Proxy with the `P2PDirectory` (FADA). Then, the `AdapterWrapper` is registered with the `WrapperRepository`. The `AdapterWrapper` for a co-ordinating BPEL compositions contains a Java representation of the WSDL interface for creating a ServiceProxy object but leaves the implementation empty as all calls for such a DBE service will be delegated to the BPELAdapter.

`AdapterWrapper` instances are retrieved from the `WrapperRepository` via a unique identifier. The identifier for normal DBE services is a concatenation of the public URL of the Servent instance (`http://hostname:2727/`) and a service identifier. We extended this identifier with "/BPEL" to differentiate composed DBE services from simple DBE services. So a BPEL composition is identified in the Servent with an identifier such as `http://hostname:2727/serviceid/BPEL` .

4.4 Execution of Coordinated BPEL Compositions

As described in Section 3.4.2, a service consumer uses the Semantic Registry to search for a DBE service. Upon finding a suitable service, a service consumer retrieves its SMID from the Semantic Registry. A SMID is used to identify a Proxy of the required service on FADA. The order of invocations that a consumer makes in order to find a Proxy object and invoke the service is shown in Figure 4.5.

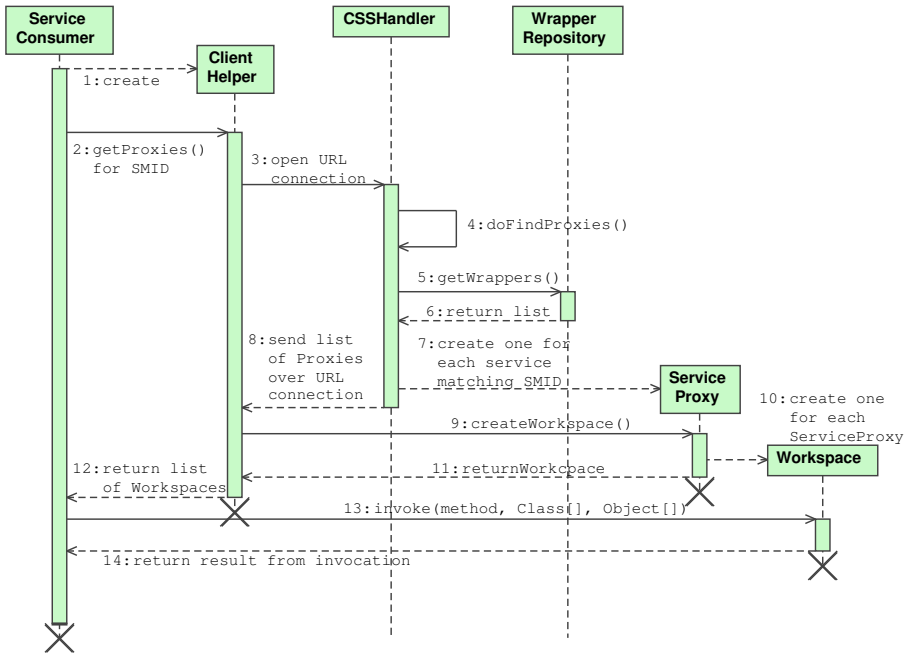


Figure 4.5: Sequence diagram of retrieving Proxies from FADA and invoking them

A consumer creates a `ClientHelper` object, which is a helper class within the Servent that encapsulates all direct contact with Servent classes for consumers.

We extended the `ClientHelper` class to request for all Proxies that match the given SMID instead of only the first matching one in order to enable the service instance fail-over mechanism. The `ClientHelper` contacts the `CSSHandler`, another class in the Servent, for retrieving Proxies. The `CSSHandler` gets a list of all available `AdapterWrapper` instances, DBE service instances within a Servent, from the `WrapperRepository` and creates a `ServiceProxy` object for each `AdapterWrapper` that matches the SMID. A list of matching `ServiceProxy` objects is then returned to the `ClientHelper`. The `ClientHelper` creates a `Workspace` object for each `ServiceProxy` object. The `Workspace` class implements the dynamic invocation interface for the Abstract Protocol Adapter (APA), offering to service consumers the method displayed in Listing 4.2 to invoke DBE services. The `Workspace` object delegates the invocation of the service consumer to the correct Servent instance.

Listing 4.2: invoke() method in Workspace class

```
1 public Object invoke(String methodName, Class[] paramTypes,
      Object[] args) {...}
```

Incoming calls for a DBE service into a Servent instance are intercepted in the `handle()` method in the `SSSHandler` class. The `SSSHandler` obtains the correct `AdapterWrapper` instance from the `WrapperRepository` via an identifier that is passed in as the `adapter` string in Listing 4.3. We modified the `SSSHandler` so that the `adapter` string may also used to decide whether the invocation must be delegated to the BPELAdapter or not. This process is shown in Listing 4.3.

Listing 4.3: handle() method in Servent

```
1 public void handle(String adapter, String pathParams,
2     ServentRequest request, ServentResponse response)
3 {
4   if (adapter.indexOf("/BPEL") > 0) { // Invocation on BPEL
        composition
5   }
6   else { // Normal DBE service is called
7   }
8   //...
9 }
```

Figure 4.6 illustrates the sequence diagram within the `SSSHandler` class upon receiving an invocation for a DBE service that represents a BPEL composition. The `SSSHandler` retrieves the correct `AdapterWrapper` instance from the `WrapperRepository` and sets up the `InvocationRequest` and `InvocationResponse` objects. The `InvocationRequest` object contains information about the method that will be invoked, an array of the parameter values and an array of the class types of the parameters such as `Holder` class types for the parameters that are returned. The invocation is then delegated to the BPELAdapter, which will be explained in detail in the following section. Upon receiving the response from the BPELAdapter, the `SSSHandler` retrieves any response values from the `InvocationResponse` object.

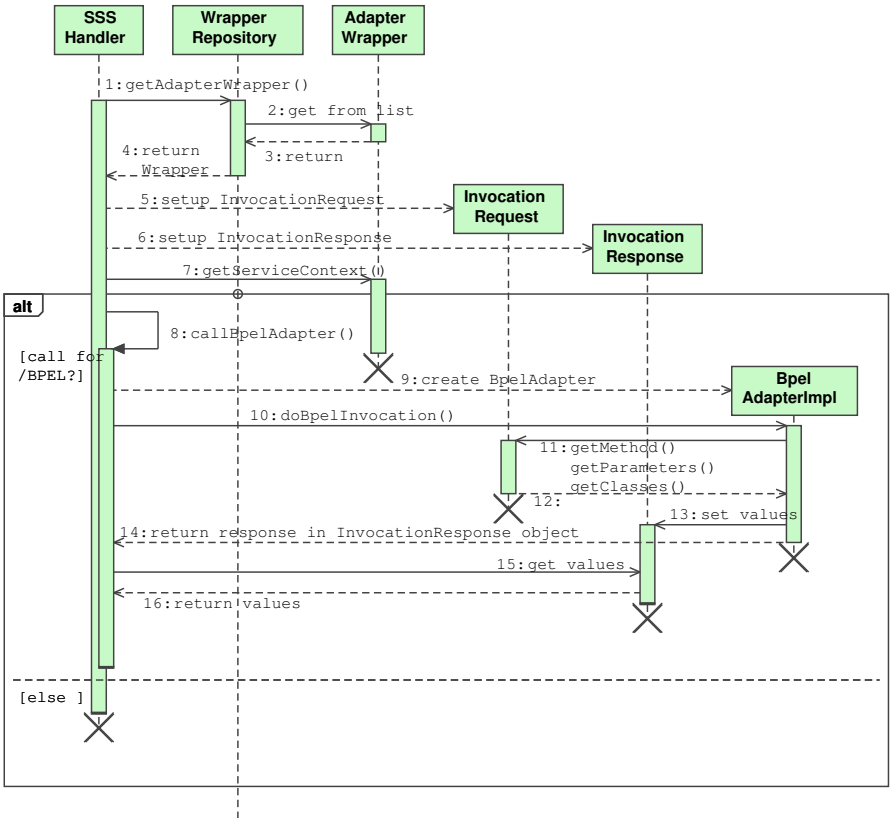


Figure 4.6: Sequence diagram of receiving an invocation in Servent

4.4.1 BPEL Adapter

The Servent delegates incoming calls for a BPEL composition to the `doBpelInvocation()` method in the BPELAdapter. An overview of what happens in this method in the BPELAdapter is given in Listing 4.4 and will be explained in the following paragraphs.

Listing 4.4: doBpelInvocation() method in BPELAdapter

```
1  public void doBpelInvocation(InvokationRequest request,
2      InvokationResponse response, ServiceContext context)
3  {
4    // Obtain values from service configuration for creating
         call...
5    // Create the Call
6    org.apache.axis.client.Call call = createCall(...);
7    // Prepare the Parameter and Holder fields
8    List[] paramCollection = createHolders(...);
9    // Invocation of SOAP call, passing in List of parameter
         objects
10   Object responseObj = call.invoke(paramCollection[0].toArray
         ());
11   // Get outputValues and add the response as first
```

```
12    List output = call.getOutputValues();
13    // Assign a value for each holder
14    for (int i = 0; i < paramCollection[1].size() && i < output
          .size(); i++) {
15      if (i == 0) {
16        response.setReturnObject(output.get(i));
17      }
18      JavaUtils.setHolderValue(paramCollection[1].get(i),
            output.get(i));
19      response.addParameter(paramCollection[1].get(i));
20    }
21  }
```

The BPELAdapter must first convert an object serialisation call into a SOAP call (see lines 3-5). In order to achieve this, a BPEL composition provides its WSDL interface to the BPELAdapter. The BPELAdapter gets all required information, such as location of the WSDL interface, service and port names from the service configuration that was supplied in the deployment descriptor (see Section 4.3.2). The `org.apache.axis.client.Service` class of Axis provides the two necessary methods to create the required `javax.xml.rpc.Call` object for a SOAP call, shown in Listing 4.5. The constructor creates a `Service` object and reads in the `javax.wsdl.Service` with the `serviceName` of the WSDL interface at the `wsdlLocation`. The `createCall()` method is then used to create the `Call` object for the required `javax.wsdl.PortType`.

Listing 4.5: Used methods in org.apache.axis.client.Service class

```
1  public class org.apache.axis.client.Service
2  implements javax.xml.rpc.Service, Serializable, Referenceable
3  {
4    public Service(String wsdlLocation, QName serviceName)
         {...}
5    public javax.xml.rpc.Call createCall(QName portName, String
         operationName) {...}
6    // ...
7  }
```

The BPELAdapter must then initialise the parameter fields with their values from the object serialisation call and prepare `javax.xml.rpc.holders.Holder` objects for any output parameters (see lines 6-7 in Listing 4.4). The `createHolders()` method in the BPELAdapter is shown in to Listing 4.6. The method returns two lists of which the first one contains the initialised parameters and the second the instantiated `Holder` objects.

Listing 4.6: Assign values to parameters and prepare Holder objects

```
1  private List[] createHolders(Object[] args, Class[]
       paramTypes)
2  {
3    // Create new parameters
4    List parameters = new ArrayList(), holders = new ArrayList
         ();
```

```
5   for (int i = 0; i < paramTypes.length; i++) {
6     // If param is a holder class
7     if (Holder.class.isAssignableFrom(paramTypes[i])) {
8       // Create the holder Object
9       Object holder = paramTypes[i].newInstance();
10      // Assign value
11      Field value = holder.getClass().getField("value");
12      value.set(holder, args[i]);
13      // Assign to parameters and to holders
14      holders.add(holder);
15      }
16    else {
17      parameters.add(args[i]);
18    }
19  }
20   return new List[] {parameters, holders};
21 }
```

The parameter objects are passed into the SOAP call (see line 8-9 in Listing 4.4). Upon receiving a response from the SOAP call, the BPELAdapter must assign the returned values correctly to the `Holder` objects that are already prepared (see line 12-19 in Listing 4.4) and add them as parameters to the `InvocationResponse` object (see line 18 in Listing 4.4).

4.4.2 Ranking Service

The Ranking Service is implemented as a simple WSDL Web service with a Java implementation. As the DBE is currently still in the implementation and testing phase, the Ranking Service had to be implemented without using the Recommender to retrieve rankings. Instead each coordinator supplies a ranking configuration file that contains the names of all coordinated BPEL compositions and a probability value for each composition. The Ranking Service is passed in the name and location of this XML-based file and reads it in. A sample of such a ranking configuration file is shown in Listing 4.7.

Listing 4.7: Contents of ranking configuration for sample service

```
1 <rankingProperties>
2    <serviceInfo>
3       <serviceName>Coordinator</serviceName>
4    </serviceInfo>
5    <rankingInformation>
6       <serviceRanking>
7         <serviceName>coordinated-process-1</serviceName>
8         <probability>0.2</probability>
9       </serviceRanking>
10      ...
11   </rankingInformation>
12 </rankingProperties>
```

The Ranking Service then calculates a random value and selects the BPEL service composition whose probability value is closest to the random value. The selected name of this BPEL service composition is then returned to the coordinator.

Listing 4.8: Ranking Service WSDL interface

```
<definitions targetNamespace="urn:rankingservice" xmlns:rank=
    "urn:rankingservice" ...>
  <message name="makeRankingRequest">
    <part name="configName" type="xsd:string"/>
    <part name="configLocation" type="xsd:string"/>
    <part name="failedService" type="xsd:string" />
  </message>
  <message name="makeRankingResponse">
    <part name="chosenService" type="xsd:string" />
  </message>
  <portType name="RankingServicePT">
    <operation name="makeRanking" parameterOrder="configName
        configLocation failedService">
      <input message="rank:makeRankingRequest" name="
          input_makeRanking"/>
      <output message="rank:makeRankingResponse" name="
          output_makeRanking"/>
    </operation>
  </portType>
  <binding .../>
  <service .../>
</definitions>
```

Listing 4.8 shows the WSDL interface of the Ranking Service. The WSDL interface of the Ranking Service offers one method `makeRanking` that has three input parameters, describing the `configName` and `configLocation` of the ranking configuration file for a specific coordinated BPEL service composition and previously failed services with `failedService`. The third parameter is empty if no BPEL service composition has failed previously. If more than one service composition has failed, the names of these are concat with "//" as a separator. The Ranking Service excludes all failed services for determining the next highest ranked BPEL service composition.

4.4.3 Customised Invoke Handler

The BPEL `invoke` activity is used to invoke internal or external Web services within a BPEL service composition. The `IAeInvokeHandler` Java interface, shown in Listing 4.9, defines the functionality that an InvokeHandler in the ActiveBPEL engine must implement. The standard InvokeHandler implementation in the ActiveBPEL engine, the `AeInvokeHandler` class, enables invocations of Web services with standard WSDL interfaces via the SOAP protocol.

Listing 4.9: IAeInvokeHandler from ActiveBPEL engine

```
public interface IAeInvokeHandler
{
  public IAeWebServiceResponse handleInvoke(IAeInvoke aInvoke
      , String aQueryData);
}
```

As described in Figure 3.9, the InvokeHandler must be customised to enable the ActiveBPEL engine to invoke DBE services as well as Web services. The customised InvokeHandler reuses the available functionality from `AeInvokeHandler` and extends it with the functionality to call on DBE services. As stated in Section 2.1.4, Proxies for DBE services are discovered and identified via their SMID and not their endpoints. Listing 4.10 shows the definition of a partner link in a PDD file (see Section 2.2.1.9) that was modified to use a DBE services:

Listing 4.10: Defining partner link of DBE service in PDD

```
<partnerLink name="coordinate1">
  <partnerRole endpointReference="static">
    <wsa:EndpointReference xmlns:coordinate1="urn:coordinate1
        ">
      <wsa:Address>smid://coordinated-process-1-SMID</
          wsa:Address>
    </wsa:EndpointReference>
  </partnerRole>
</partnerLink>
```

The value of the `Address` element starts with `smid://` to indicate that this partner link references a DBE service. As Listing 4.11 illustrates, the SMID can be easily retrieved from the PDD file and enables the customised InvokeHandler to decide whether the following invocation will call a DBE service or a Web service.

Listing 4.11: Decision if partner link defines DBE service or Web service

```
public IAeWebServiceResponse handleInvoke(IAeInvoke
    aInvokeQueueObject, String aQueryData)
{
  IAeWebServiceEndpointReference endpointReference =
      getEndpointReference(aInvokeQueueObject);
  String address = endpointReference.getAddress();
  if (address.startsWith("smid://")) { // have DBE service
  }
  else { // have normal Web service
  }
  //...
}
```

The customised InvokeHandler retrieves any required Proxy objects invoking on a DBE service in the same way as it is described at the beginning of Section 4.4 for all service consumers. Upon receiving one or more `Workspace` objects, the customised InvokeHandler must set up an object serialisation call. Although a `Workspace` object will normally contain a graphical user interface (GUI) that a service consumer can use straight away, the InvokeHandler cannot use a GUI as BPEL service compositions are executed without any user input. Therefore, the responsibility to set up the array of parameter values and class types lies by the InvokeHandler. Unlike SOAP calls, an object serialisation cannot be created automatically from the available WSDL interface as the correct Java classes are required. The InvokeHandler uses the helper class `ApaUtils` that contains the standard WSDL types and their corresponding

Java class representations to create the required class types for the object serialisation call. Listing 4.12 shows an extract of `ApaUtils`:

Listing 4.12: setting up Java object for xml type

```
1  private static HashMap constructorMap = new HashMap();
2  static
3  {
4    constructorMap.put("string", new String());
5    constructorMap.put("boolean", new Boolean(false));
6  }
7  public static Object getConstructorForType(String paramType)
8  {
9    return (Object) constructorMap.get(paramType);
10 }
```

Listing 4.13 shows the setting up of the `Object` and `Class` arrays for the object serialisation call with the correct data and class types. Lines 12-17 in Listing 4.13 show the initialisation of parameters with the mode IN and lines 18-23 the initialisation of the Holder objects for parameters with the mode OUT. The Java type that is used in line 13 and 19 was created with `ApaUtils`.

Listing 4.13: Setting up the object serialisation call in Invoke Handler

```
1  private void invokeApaRpcCall(IAeInvoke aInvokeQueueObject,
       AeInvokeResponse aResponse, Operation oper, Call call,
       Workspace workspace)
2  {
3    String operationName = aInvokeQueueObject.getOperation();
4    Map messageData = aInvokeQueueObject.getInputMessageData().
         getMessageData();
5    // Initialising the arrays
6    int inParamSize = call.getOperation().getAllInParams().size
         ();
7    int outParamSize = call.getOperation().getAllOutParams().
         size();
8    Class[] paraClasses = new Class[inParamSize + outParamSize
         ];
9    Object[] paraObjects = new Object[inParamSize +
         outParamSize];
10   int i=0;
11   for (Iterator iter = call.getOperation().getAllInParams().
         iterator(); iter.hasNext();) {
12     ParameterDesc part = (ParameterDesc) iter.next();
13     paraClasses[i] = part.getJavaType();
14     paraObjects[i] = messageData.get(part.getName());
15     i++;
16   }
17   for (Iterator iter = call.getOperation().getAllOutParams().
         iterator(); iter.hasNext();) {
18     ParameterDesc part = (ParameterDesc) iter.next();
19     paraClasses[i] = part.getJavaType();
20     paraObjects[i] = ApaUtils.getConstructorForType(part.
           getJavaType().getName());
```

```
21      i++;
22    }
23    // Invoke on workspace object
24    Object[] returnObj = (Object[]) workspace.invoke(
          operationName, paraClasses, paraObjects);
25    //...
26  }
```

Upon receiving the response from the invocation on the `Workspace` object, the InvokeHandler must set up the `AeInvokeResponse` object for the ActiveBPEL engine. The Java code for this is shown in Listing 4.14. For each output parameter, the returned value is retrieved (line 12) and added to the output message (line 13).

Listing 4.14: Setting up the response object for ActiveBPEL engine

```
1  private void invokeApaRpcCall(IAeInvoke aInvokeQueueObject,
       AeInvokeResponse aResponse, Operation oper, Call call,
       Workspace workspace)
2  {
3    //...
4    // Invoke on workspace object
5    Object[] returnObj=(Object[]) workspace.invoke(operationName
         ,paraClasses,paraObjects);
6    // Setup response object
7    QName outMsgQName = oper.getOutput().getMessage().getQName
         ();
8    AeWebServiceMessageData outputMsg = new
         AeWebServiceMessageData(outMsgQName);
9    if (outParamSize > 0) {
10     for (int j = 0; j < outParamSize; j++) {
11       ParameterDesc para=(ParameterDesc)call.getOperation().
             getAllOutParams().get(j);
12       Object output = JavaUtils.getHolderValue(paraObjects[j+
             inParamSize]);
13       outputMsg.setData(para.getName(), output);
14     }
15   }
16   // Return the message to the awaiting callback
17   aResponse.setMessageData(outputMsg);
18 }
```

4.4.3.1 Complex Types

Java class types for complex types are problematic as they cannot be created automatically. Therefore the deployment descriptor was extended (see Section 4.3.2) to contain the name of a complex type Java class and an entry was added to the Proxy object. The problem is partitioned into two subproblems, where the first problem is to obtain an instance of the required class. The second problem is how to instantiate the object with the correct constructor and add the values for the member variables that were passed from the ActiveBPEL engine. Listing 4.15 illustrates how the required Class type is obtained from the remote code-base of a DBE service.

Listing 4.15: Obtaining the class of complex type from remote service

```
1 public Class getComplexTypeClass(String complexName, Map
      serviceConfig)
2 {
3   String codebase = (String) serviceConfig.get("
        dbe_server_jvm_codebase");
4   URL[] urls = new URL[] { new URL(codebase) };
5   ClassLoader loader = this.getClass().getClassLoader();
6   URLClassLoader urlLoad = new URLClassLoader(urls, loader);
7   return urlLoad.loadClass(complexName);
8 }
```

Listing 4.16 shows how to create an object for the complex type and assign the correct values to each member variable.

Listing 4.16: Create object and assign correct values to member variables of the object

```
1 public Object getComplexTypeObject(Class complexClass,
      Document data)
2 {
3   List paramsList = new ArrayList();
4   Node parent = data.getChildNodes().item(0);
5   NodeList cParams = parent.getChildNodes();
6
7   Constructor[] constructors = complexClass.getConstructors()
        ;
8   Constructor correctConstructor = null;
9   // Getting constructor
10  for (int i=0; i<constructors.length && correctConstructor
        == null; i++) {
11    if(constructors[i].getParameterTypes().length == cParams.
          getLength()) {
12      correctConstructor = constructors[i];
13    }
14  }
15  // Getting params
16  Class[] pClasses = correctConstructor.getParameterTypes();
17  for (int j=0; j<cParams.getLength(); j++) {
18    Node param = cParams.item(j);
19    // Creates object with classname as 1st param, value as 2
          nd param
20    Object pObject = ApaUtils.initObjectWithValue(pClasses[j
          ].getName(), param.getFirstChild().getNodeValue());
21    paramsList.add(pObject);
22  }
23  return correctConstructor.newInstance(paramsList.toArray())
        ;
24 }
```

In lines 7-14, the correct constructor for the required object that has the same number of parameters as the complex type parts is obtained. For each parameter of the Class constructor, we obtain the Java class type and the value and create the corresponding object (lines 15-23). Listing 4.17 illustrates the method that is used in line 20

in Listing 4.16 to create the required Java object with a given value. Finally, the Java object for a complex type can be created in line 24 of Listing 4.16.

Listing 4.17: setting up Java object with value

```
1  public static Object initObjectWithValue(String classType,
       String value)
2  {
3    if (classType.equals("java.lang.Boolean")) {
4      return new Boolean (value);
5    }
6    else if (classType.equals("java.lang.String")) {
7      return new String (value);
8    }
9    // ...
10 }
```

This solution has been tested and currently works with complex types that only contain simple parts. The nesting of complex types has not been implemented.

4.4.4 Failure Handling Mechanisms

We defined two failure handling mechanisms for the coordination architecture in Section 3.4.3: service instance fail-over and service composition fail-over. The customised Invoke-Handler is responsible for implementing the first mechanism while the coordinator enables the second mechanism.

The customised InvokeHandler retrieves `Workspace` objects for all `ServiceProxy` objects that map to the same SMID from FADA and uses one of the returned `Workspace` objects for executing the invocation. The call for invoking on the `Workspace` object is surrounded by a try-catch block, as shown in Listing 4.18.

Listing 4.18: Failover in customized Invoke Handler

```
1  //...
2  List workspaces = (List) css.getProxies(null, entries);
3  boolean success = false;
4  int tries = 0;
5  while (tries < workspaces.size() && !success) {
6    try {
7      Workspace workspace = (Workspace) workspaces.get(tries);
8      invokeApaRpcCall(aInvokeQueueObject,response,operation,
           call,workspace);
9      success = true;
10   }
11   catch (Exception e) {} // try failover to another Workspace
12   tries++;
13 }
```

If the selected `Workspace` object could not be used or threw an exception and there is more than one `Workspace` object available, the InvokeHandler fails over to another `Workspace` object.

If no other `Workspace` object is available or all available ones have failed, the InvokeHandler throws an exception and the BPEL service composition will fail. In

this case, we apply the service composition fail-over mechanism. `Invoke` activities in a BPEL service composition can define exception handling that enables the BPEL process to inform the service consumer or in our case the coordinator of the service failure and finish processing gracefully (see Section 2.2.1.7).

The coordinator checks whether all BPEL service compositions have failed in the following way:

Listing 4.19: Using all available service compositions

```
1  <assign>
2    <copy>
3      <from expression=" '5' "/>
4      <to variable="max_tries"/> <!-- type="xsd:integer"-->
5    </copy>
6    <copy>
7      <from expression=" '1' "/>
8      <to variable="tries"/> <!-- type="xsd:integer"-->
9    </copy>
10 </assign>
11 <while condition="bpws:getVariableData('v_clientResponse','
       successful')!=true() and bpws:getVariableData('tries')&lt
       ;=bpws:getVariableData('max_tries')">
12   <sequence>
13     <!-- get ranking and invoke on selected booking-1x -->
14     <assign>
15       <copy>
16         <from expression="bpws:getVariableData('tries')+1"/>
17         <to variable="tries"/>
18       </copy>
19     </assign>
20   </sequence>
21 </while>
```

Assuming that the coordinator has five BPEL service compositions to coordinate, this number is kept in `max_tries` while `tries` is counted up from one to maximal five. A while loop checks whether the previously selected BPEL service composition was successful and if all available BPEL service compositions were used (line 11 in Listing 4.19). After using a BPEL service composition the variable `tries` is incremented by one (lines 14-19). Prior to invoking the Ranking Service again, the coordinator sets the `failedService` parameter up again. The Listing 4.20 assumes that the selected service is kept in `v_service`. The `failedService` variable is then a concatenation of all previously failed service compositions separated by "//" with `v_service`:

Listing 4.20: Creating string for failed services to pass to RankingService

```
1  <assign name="SetUp_FailedServices_Array">
2    <copy>
3      <from expression="concat(bpws:getVariableData('
           v_rankRequest', 'failedService'),
           bpws:getVariableData('v_service'), '//')" />
4      <to part="failedService" variable="v_rankRequest"/>
5    </copy>
```

```
6 </assign>
```

4.4.5 Transitioning between Different Service Types

As described in Section 3.4.4, the transitioning between different service types can either include simple XPath based transformations or the invocation of another service to retrieve required service types. As this task is performed in the coordinator that must be specified individually by a service provider for each coordinated service composition, no implementation was required at this stage of the deliverable.

4.5 Summary

In this chapter we presented our implementation of the coordination architecture. An introduction to the development environment was given in Section 4.1.

Section 4.2 detailed technical aspects of the coordination architecture along with each of its components. The deployment of coordinated BPEL service compositions was presented in Section 4.3. The DAR file that is used to deploy DBE services to the Servent was extended by a `service-composition` folder that contains all service composition relevant information (Section 4.3.1). Also, the deployment descriptor had to be enhanced to supply the Servent with information about the BPEL service compositions (Section 4.3.2). The DeployerImpl (Section 4.3.3) was updated to use this information to deploy BPEL service compositions in the correct order to the ActiveBPEL engine. A BPEL specific suffix was added to the key that the Servent uses for correlating incoming requests with deployed service instances.

Section 4.4 depicted the consumption of coordinated BPEL service compositions. Incoming requests for BPEL service compositions to the Servent are delegated to the BPELAdapter. The BPELAdapter transforms the object serialisation call into a SOAP call (Section 4.4.1) and invokes the BPEL process. After a BPEL process has finished the BPELAdapter transforms the result back into the correct format for DBE services.

At execution-time, the coordinator uses the Ranking Service to retrieve the highest ranked BPEL service composition. The Ranking Service reads in a ranking configuration file and randomly selects one BPEL service composition (Section 4.4.2). If a BPEL service composition fails, the Ranking Service is contacted again. It excludes the failed BPEL service composition from the list of available services and repeats the selection process.

The customised InvokeHandler (Section 4.4.3) has to support DBE services as well as standard Web services and therefore extends the `AeInvokeHandler` class in the ActiveBPEL engine. The InvokeHandler must first decides whether the invocation concerns a DBE service or a standard Web service. To invoke a DBE service, a Proxy for the supplied SMID must be discovered and downloaded. Then an object serialisation call for this Proxy must be set up and invoked. The returned result must be assigned to the correct variables in the BPEL process.

Section 4.4.4 described the implementation of the failure handling mechanisms that the coordination architecture uses to provide increased failure resilience. The fail-over to another service instance of the same service type required an extension of the `ClientHelper` class to return a set of Proxies instead of only one. Within the InvokeHandler, an error in one service instance initiated the fail-over to another service instance if more than one is available. The second failure handling mechanism required

work in the BPEL definition of the coordinator. The coordinator was implemented to repeat the process to select one BPEL service composition with the Ranking Service and delegate the request to it until either one invocation is successful or all BPEL service compositions have failed.

Chapter 5

Evaluation

This chapter reports on the evaluation of the coordination architecture in the DBE based on the requirements that we presented in Section 3.1. The coordination architecture is evaluated with respect to service composition reliability (Section 5.3), failure resilience Section 5.4 and ease of use for consumers (Section 5.5). We use five clients that invoke concurrently on the coordination architecture to measure success and failure rates for the clients. The same test is carried out for simple service compositions to compare the results. Also, the success rate for a service instance fail-over mechanism is evaluated. Finally, we measure the performance for clients from the start of an invocation until a reply is received to rate the deterioation of the coordination architecture on the experience delay by consumers.

5.1 Meeting the Requirements

In Section 3.1, a set of requirements were identified for the coordination architecture. The following list depicts each individual requirement and outlines how they were met.

1. Open-source environment and standards-based:
   The implementation of the coordination architecture was achieved by using open-source projects, such as the ActiveBPEL engine to execute BPEL processes. Where possible, standards or emerging standards such as BPEL, WSDL and SOAP are used. The coordination architecture uses BPEL-compliant constructs to implement the coordinator.

2. Increased service composition reliability:
   In order to achieve an increased service composition reliability, several BPEL service composition are deployed that offer equivalent functionality. The coordinator can choose the highest-ranking one and delegate the incoming request to it. If one BPEL service composition fails, the coordinator can redirect the request to the next highest BPEL service composition.

3. Failure resilience:
   As several redundant BPEL service compositions are deployed that can replace each other, the failure resilience is increased. Also, the fact that several DBE service instances can implement the same service type and be discovered via the same SMID increases the failure resilience.

4. Ease of use for consumers:
   The ease of use for service consumers is increased by offering an increased service composition reliability. If a service composition fails, a service consumer would have to try and find another service that suits his requirements. A service provider that uses the coordination architecture can reduce the need for this to a minimum while keeping consumers unaware of most service failures.

5.2 Evaluation Scenario

In order to evaluate the coordination architecture, a number of test services were created. Figure 5.1 illustrates the scenario for evaluating the architecture.

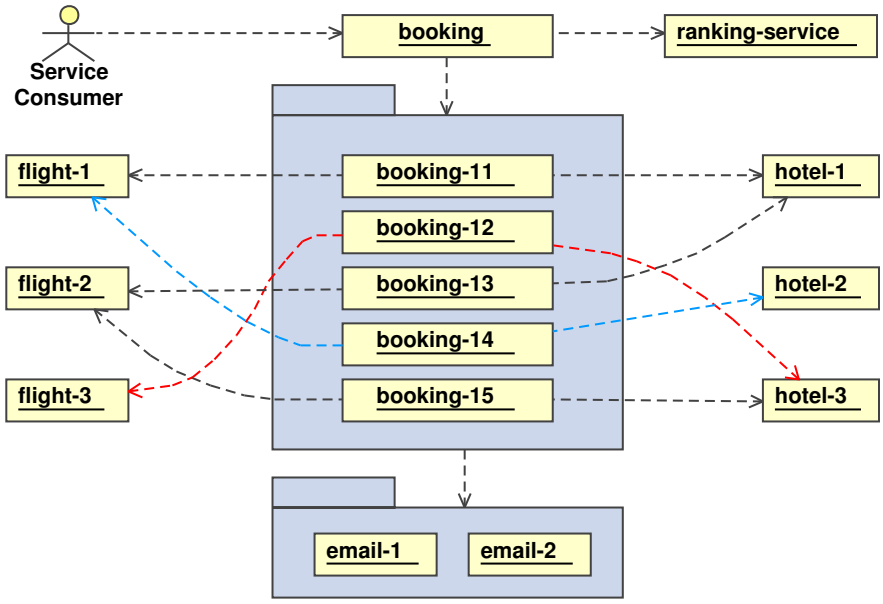


Figure 5.1: Service usage for evaluation

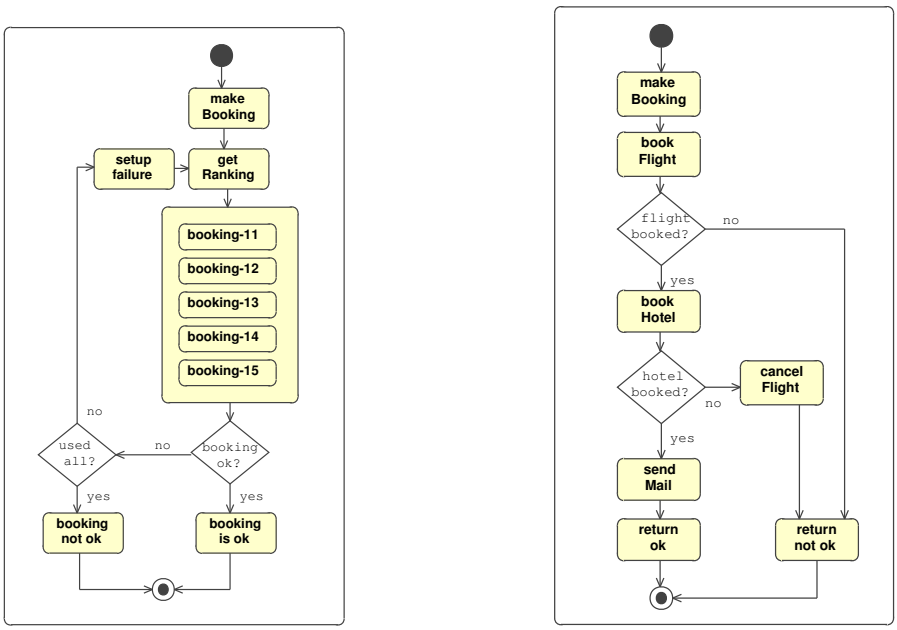
The `booking` service is the coordinator that coordinates the execution for five other BPEL service compositions `booking-11`, `booking-12`, `booking-13`, `booking-14` and `booking-15`. Each BPEL service composition depends upon three DBE services, a flight and hotel booking service and an email service that is used to send a receipt to a service consumer after a successful booking. Available DBE services are three different flight booking services `flight-1`, `flight-2` and `flight-3` and three different hotel booking services `hotel-1`, `hotel-2` and `hotel-3`. The `flight-2` and `hotel-2` service differ from the other services in using complex types. The email services, `email-1` and `email-2`, are instances of the same service type and use the same SMID so that a fail-over to another service instance can be shown.

For the evaluation scenario it is assumed that failures during using a flight or hotel booking service result in a complete failure of a BPEL service composition. A failure in a email service does not affect the booking process and can be considered a partial

failure of a BPEL service composition as a service consumer will still receive the booking reference. The evaluation results show how many times a consumer did not receive an email receipt.

As illustrated in Figure 5.2 (a), the coordinator receives a request from a service consumer and retrieves a ranking from the ranking service. The consumer's request is then delegated to the selected BPEL service composition. When the selected BPEL service composition returns to the coordinator it decides whether the booking was successful or not. If the booking was successful, the service consumer is returned the booking information. Otherwise, the coordinator checks whether all available BPEL service compositions where used or not. If not all BPEL service compositions have failed, a new ranking is retrieved, excluding previously failed BPEL service compositions, and the execution is continued with the next selected BPEL service composition. If all available BPEL service compositions have failed, a negative reply is sent to the service consumer.

Figure 5.2 (b) shows the BPEL process for a coordinated BPEL service composition. The BPEL process attempts to book a flight. If a flight can be booked, the BPEL process tries to book a room in a hotel as well. If this booking succeeds, the BPEL process sends a receipt of the booking to the service consumer via an email and returns to the coordinator. Otherwise, the BPEL process cancels the previously booked flight and returns to the coordinator with the result that the booking failed.



(a) coordinator booking service

(b) individual booking service sample

Figure 5.2: State diagrams for coordinator and coordinated service composition

Each scenario is run in the following way: Five service consumers are each con-

currently invoking 500 calls on the coordinated booking service. To compare the result with not coordinated service compositions, each individual booking service is tested under the same constraints. Table 5.1 gives an overview of the evaluation scenarios.

scenario	tested composition	service failures %	clients/calls	composition type
A	booking	5, 10, 20, 40	5 / 500	coordinated
B	booking-11	5, 10, 20, 40	5 / 500	non-coordinated
C	booking-12	5, 10, 20, 40	5 / 500	non-coordinated
D	booking-13	5, 10, 20, 40	5 / 500	non-coordinated
E	booking-14	5, 10, 20, 40	5 / 500	non-coordinated
F	booking-15	5, 10, 20, 40	5 / 500	non-coordinated

Table 5.1: Description of the Evaluation Scenarios

5.2.1 Setting up the Service Failure Rates

The described evaluation scenario was run for the four service failure rates 5%, 10%, 20% and 40%. To achieve a random failure within a service the following code was used:

Listing 5.1: Random failure of service with a percentage

```
1 Random rnd = new Random(new Date().getTime());
2 int randomVal = rnd.nextInt(100); // returns 0 >= randomVal
      < 100
3 if (randomVal < 5) { // FAILURE–RATE = 5%
4   // fail service...
5 }
6 // continue with normal execution...
```

The difference between the real service failure rate and the one that is set is shown in Appendix B. The values were taken from evaluation scenario A, running the test clients for a coordinated BPEL service composition. The result shows that the random failure in Listing 5.1 achieves the aimed at service failure rate well.

5.2.2 Measure Points

Specific measure points are defined to evaluate and compare the different scenarios. Figure 5.3 gives an overview of these measure points.

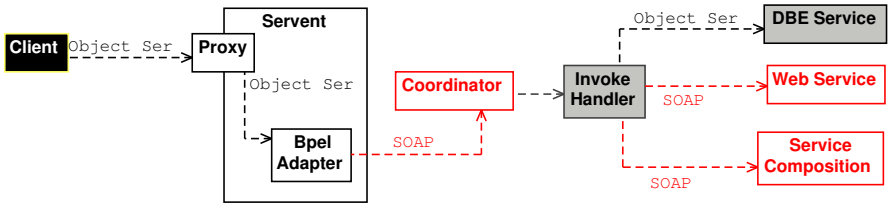


Figure 5.3: Measure points at test-time for evaluation

The most important measure point is the Client (black box). Within each Client the success and failure rates are counted to make a statement about the service composition reliability with and without using the coordination architecture. Also, the duration of each invocation is measured in the Clients by logging the start and end of an invocation in conjunction with the success/failure status. Log4J [25] was used for the logging in the Clients. The accuracy that can be achieved depends on the performance of the host computer CPU and the accuracy of Log4J. The duration is important to make a statement about the performance deterioration of Clients that use the coordination architecture.

Two other measure points are configured in the InvokeHandler and each DBE service (grey boxes). These points only measure the success and failure rates because the performance proved to be too diminutive to measure. The success and failure status of each invocation is stored in a HSQLDB database [31]. Within the DBE services, the measure point can be used to prove that each service is failing with the set failure rate. The InvokeHandler knows about all invocations that any BPEL service composition invokes. Therefore, the measure point allows to make a statement about the evenly distribution of incoming requests to all coordinated service compositions. Also, the increased usage of the coordination architecture with the rising service failure rates can be proved.

5.2.3 Ranking Configuration

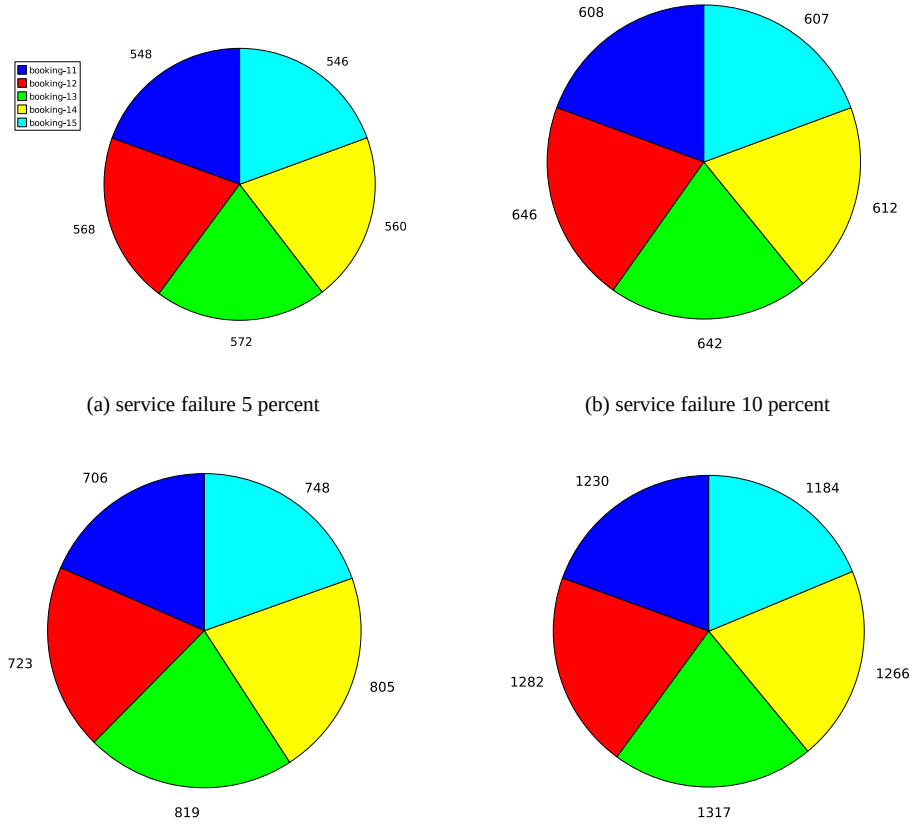
The ranking configuration that is used for the evaluation is displayed in Listing 5.2.

Listing 5.2: Ranking configuration for evaluation

```
<rankingProperties>
  <rankingInformation>
    <serviceRanking>
      <serviceName>booking-11</serviceName>
      <probability>0.1</probability>
    </serviceRanking>
    <serviceRanking>
      <serviceName>booking-12</serviceName>
      <probability>0.3</probability>
    </serviceRanking>
    <serviceRanking>
      <serviceName>booking-13</serviceName>
      <probability>0.5</probability>
    </serviceRanking>
    <serviceRanking>
      <serviceName>booking-14</serviceName>
      <probability>0.7</probability>
    </serviceRanking>
    <serviceRanking>
      <serviceName>booking-15</serviceName>
      <probability>0.9</probability>
    </serviceRanking>
  </rankingInformation>
</rankingProperties>
```

The chosen probability values enable to select all coordinated BPEL service composi-

tions with an equal chance. The graphics in Figure 5.4 show that all compositions are equally chosen by the coordinator. Another characteristic is that the overall usage increases with higher service failures as the coordinator has to fail-over to another service composition more often. Detailed values can be found in Table C.2.



(a) service failure 5 percent

(b) service failure 10 percent

(c) service failure 20 percent

(d) service failure 40 percent

Figure 5.4: Usage of coordinated service compositions for 5, 10, 20 and 40 percent service failures

The usage of the Ranking Service is shown in Figure 5.5 (see Table C.1 for detailed values). The minimum number that the Ranking Service is going to be called is 2500 times, 500 for each client. The usage of the Ranking Service increases linearly with higher service failures.

5.2.4 Evaluation Environment

All evaluation experiments presented in this deliverable have been conducted by implementing the selected scenarios as prototypical applications. The booking service compositions and the coordinator have been implemented BPEL-compliant. Each of the hotel, flight and email services is realised as a DBE service and discovered via their SMID. As FADA currently has scalability issues, the services had to be hosted on one

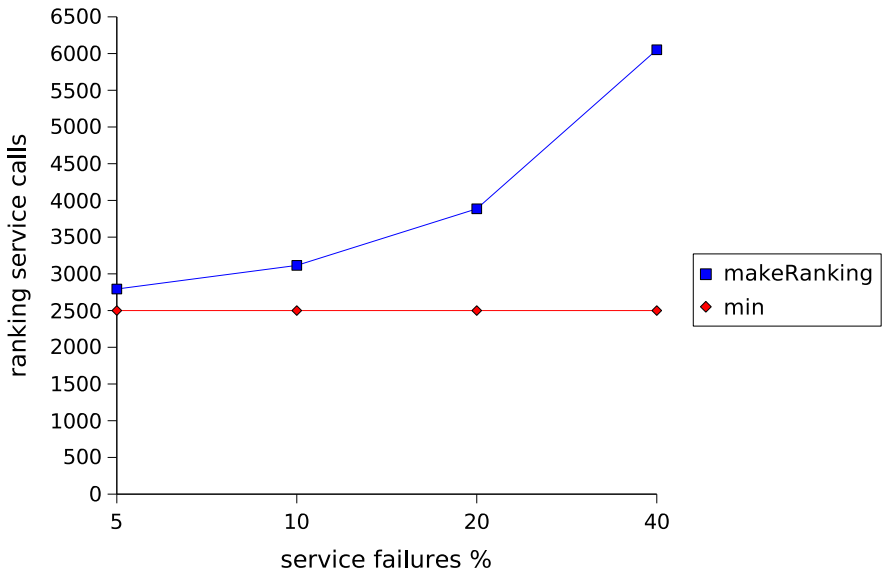


Figure 5.5: Ranking service usage

computer only, running the Linux (Ubuntu) operating system on a 3GHz Intel Pentium IV processor.

5.3 Service Composition Reliability

In order to achieve increased service composition reliability, our approach may deploy several BPEL service compositions with equivalent functionality. The coordinator can choose the highest-ranking one and delegate the incoming request to this service. If one BPEL service composition fails, the coordinator can redirect a request to the next highest BPEL service composition. The following subsections illustrate how the service composition reliability of coordinated and non-coordinated service compositions is affected by different service failure rates. Detailed results of successes, failures and client failures % for individual clients can be found in Appendix A. This section gives a summary of all five clients for each evaluation scenario.

5.3.1 5% Service Failure

This section displays the results for a 5% service failure rate for all used services. Table 5.2 gives an overview of the success rate for clients in this scenario. The presented values are a sum of the results for the individual clients.

	service failures %	success	failures	total calls	client failures %
Coord	5	2500	0	2500	0
Book-11	5	2264	236	2500	9.44
Book-12	5	264	2236	2500	10.56
Book-13	5	255	2245	2500	10.2
Book-14	5	244	2256	2500	9.76
Book-15	5	265	2235	2500	10.6

Table 5.2: Client results with and without using architecture for 5 % service failure

The client failure rate is calculated with the following equation:

EQ-1: $clientFailures = (failures * 100)/totalCalls$

As it can be seen, the success rate of clients that use the coordination architecture is a 100%. Despite service failures, the coordination architecture can guarantee full service composition reliability whereas all other service compositions exhibit a failure rate of 9.44% to 10.6%.

5.3.2 10% Service Failure

This section displays the results for a 10% service failure rate for all used services. Table 5.3 gives an overview of the success rate for clients in this scenario.

	service failures %	success	failures	total calls	client failures %
Coord	10	2500	0	2500	0
Book-11	10	2027	473	2500	18.92
Book-12	10	2029	471	2500	18.84
Book-13	10	2015	485	2500	19.4
Book-14	10	2047	453	2500	18.12
Book-15	10	2023	477	2500	19.08

Table 5.3: Client results with and without using architecture for 10 % service failure

As it can be seen, the success rate of clients that use the coordination architecture is still 100%. Despite service failures, the coordination architecture can guarantee full service composition reliability whereas all other service compositions only succeed with a 81% chance.

5.3.3 20% Service Failure

This section displays the results for a 20% service failure rate for all used services. Table 5.4 gives an overview of the success rate for clients in this scenario.

	service failures %	success	failures	total calls	client failures %
Coord	20	2485	15	2500	0.6
Book-11	20	1624	876	2500	35.04
Book-12	20	1571	929	2500	37.16
Book-13	20	1583	917	2500	36.68
Book-14	20	1631	869	2500	34.76
Book-15	20	1587	913	2500	36.52

Table 5.4: Client results with and without using architecture for 20 % service failure

As it can be seen, the success rate of clients that use the coordination architecture is 99%. Despite service failures, the coordination architecture can guarantee nearly full service composition reliability whereas all other service compositions only succeed with a 65% chance.

5.3.4 40% Service Failure

This section displays the results for a 40% service failure rate for all used services. Table 5.5 gives an overview of the success rate for clients in this scenario.

	service failures %	success	failures	total calls	client failures %
Coord	40	2273	227	2500	9.08
Book-11	40	895	1605	2500	64.2
Book-12	40	870	1630	2500	65.2
Book-13	40	894	1606	2500	64.24
Book-14	40	940	1560	2500	62.4
Book-15	40	882	1618	2500	64.72

Table 5.5: Client results with and without using architecture for 40 % service failure

As it can be seen in success rate of clients that use the coordination architecture is 91%. Despite massive service failures, the coordination architecture can guarantee a good failure resilience whereas all other service compositions only succeed with a 36% chance. The result for the coordinated service composition is actually better then the achievement for non-coordinated service compositions at a 5% service failure rate.

5.4 Failure Resilience

As we stated in Section 3.1, one of the main goals of the coordination architecture is to achieve better failure resilience. This was achieved by the two-step failure handling that was introduced in Section 3.4.3. The results of successes, failures and client failures % for individual clients can be found in Appendix A. This section gives a summary of all five clients for each evaluation scenario.

The failure rate has to be considered for the two failure handling mechanisms that were introduced in Figure 3.11. The first mechanism can be used if one service type is implemented by more then one service instance. The results for this mechanism will be displayed in Section 5.4.1.

The second mechanism is used within the coordinator to fail-over to another BPEL service composition if a selected BPEL service composition fails. The results for this mechanism will be displayed in Section 5.4.2.

5.4.1 Fail-over to another Service Instance

The email service type is implemented by the two service instances `email-1` and `email-2`. This enables the fail-over to the other service instance if one fails, the first failure handling mechanism that was described in Section 3.4.3. The expected failure rate for the email service type is thereby reduced from the general service failure rate and can be calculated with the following equation:

EQ-2: $expectedFailure = serviceFailure * serviceFailure * 100$

The real failure rate for the email service is calculated with the following equation:

EQ-3: $realFailure = (noReceipt * 100)/clientSuccess$

The fail-over to another service instance is used by coordinated and non-coordinated service compositions. As the results will show, this failure handling mechanism performs better for a low service failure rate. This can also be proven mathematically with the following equation:

EQ4: $improvement = serviceFailure/expectedFailure$

The following subsections will provide the results for the different service failure rates.

5.4.1.1 5% Service Failure

As the results in Table 5.6 show, the failure rate of a service that uses fail-over to other service instances offer a 20 times better failure resilience for the email service type, e.g. a service failure rate of 5% is reduced to 0.25%.

	service failures %	email success	client success	no receipt	no receipt (exp) %	no receipt (real) %
Coord	5	2495	2500	5	0.25	0.2
Book-11	5	2257	2264	7	0.25	0.31
Book-12	5	2235	2236	1	0.25	0.04
Book-13	5	2237	2245	8	0.25	0.36
Book-14	5	2251	2256	5	0.25	0.22
Book-15	5	2233	2235	2	0.25	0.09

Table 5.6: No receipt sent for 5 % service failure

5.4.1.2 10% Service Failure

As the results in Table 5.7 show, the failure rate of a service that uses fail-over to other service instances offer a 10 times better failure resilience for the email service type, e.g. a service failure rate of 10% is reduced to 1%.

	service failures %	email success	client success	no receipt	no receipt (exp) %	no receipt (real) %
Coord	10	2476	2500	24	1	0.96
Book-11	10	2005	2027	22	1	1.09
Book-12	10	2000	2029	29	1	1.43
Book-13	10	1991	2015	24	1	1.19
Book-14	10	2014	2047	33	1	1.61
Book-15	10	2004	2023	19	1	0.94

Table 5.7: No receipt sent for 10 % service failure

5.4.1.3 20% Service Failure

As the results in Table 5.8 show, the failure rate of a service that uses fail-over to other service instances offer a five times better failure resilience for the email service type, e.g. a service failure rate of 20% is reduced to 4%.

	service failures %	email success	client success	no receipt	no receipt (exp) %	no receipt (real) %
Coord	20	2402	2485	83	4	4.5
Book-11	20	1551	1624	73	4	3.34
Book-12	20	1518	1571	53	4	3.37
Book-13	20	1533	1583	50	4	3.16
Book-14	20	1533	1631	98	4	6.01
Book-15	20	1515	1587	72	4	4.54

Table 5.8: No receipt sent for 20 % service failure

5.4.1.4 40% Service Failure

As the results in Table 5.9 show, the failure rate of a service that uses fail-over to other service instances offer a two and a half times better failure resilience for the email service type, e.g. a service failure rate of 40% is reduced to 16%.

	service failures %	email success	client success	no receipt	no receipt (exp) %	no receipt (real) %
Coord	40	1984	2273	289	16	12.71
Book-11	40	733	895	162	16	18.1
Book-12	40	725	870	145	16	16.67
Book-13	40	747	894	147	16	16.44
Book-14	40	747	940	193	16	20.53
Book-15	40	733	882	149	16	16.89

Table 5.9: No receipt sent for 40 % service failure

5.4.2 Fail-over to another Service Composition

The second failure handling mechanism that the coordination architecture uses is the fail-over to another service composition. As several redundant BPEL service compositions are available this approach offers a much better failure resilience. Figure 5.6 gives an overview of the failure resilience for clients that use the coordination architecture and those that do not. The graphic is based on the results that were presented in Section 5.3.

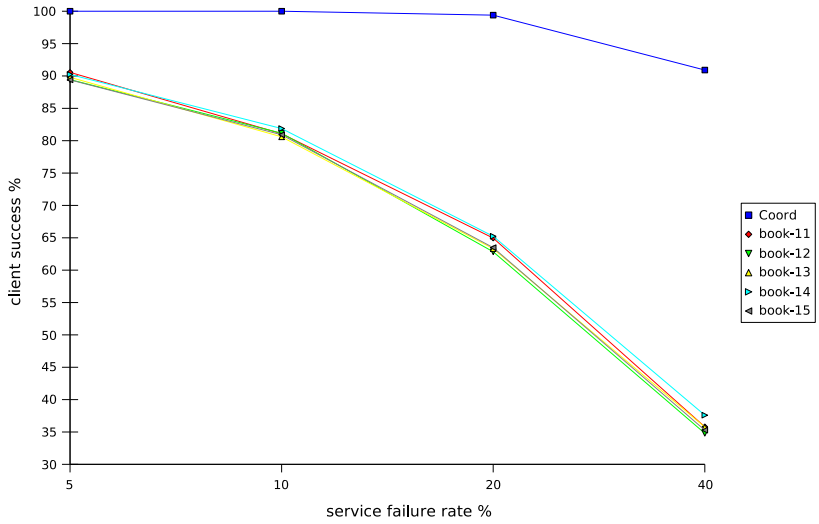


Figure 5.6: Client success with and without using coordination architecture

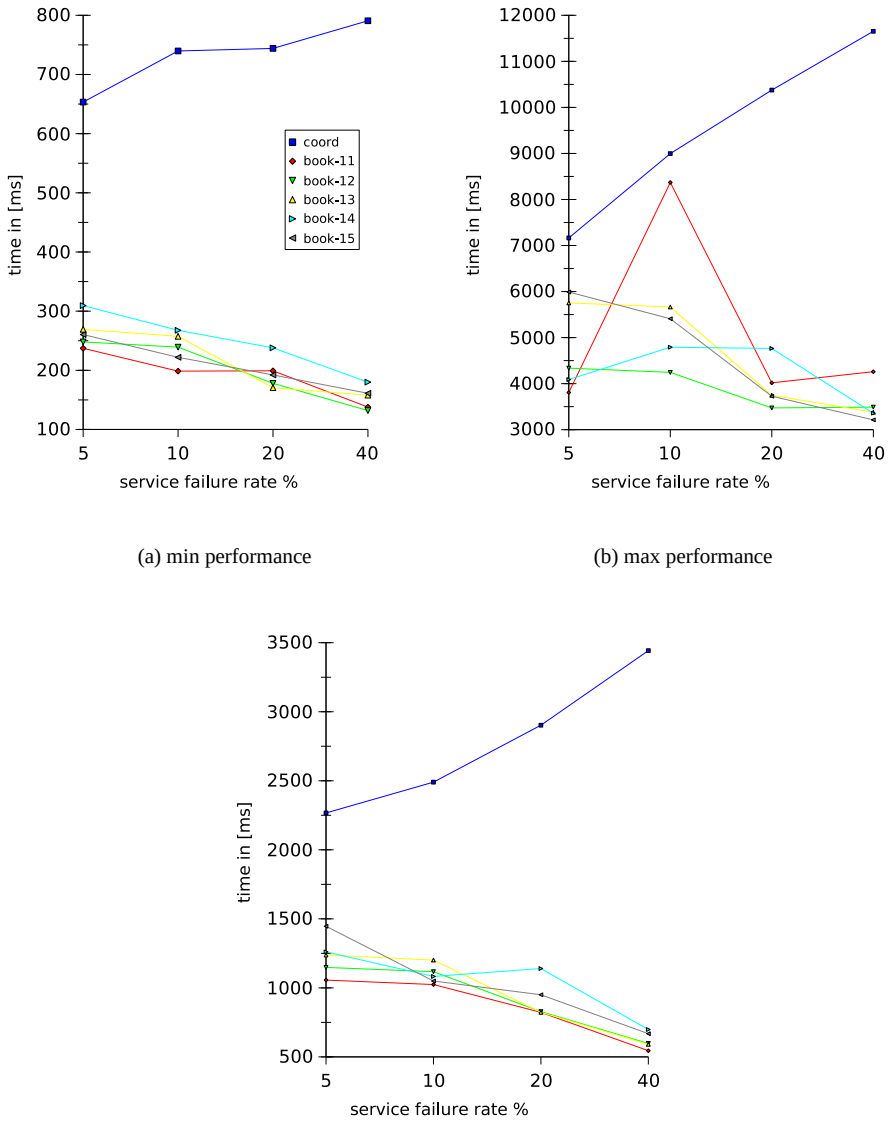
It can be seen that the coordination architecture improves the failure resilience significantly. For 5% and 10% service failures, the coordination architecture can completely hide any failures from a service consumer. Even with 20% service failures the coordination architecture achieves a nearly full failure resilience. The result for 40% service failures is interesting as the coordination architecture still achieves a better failure resilience in this unstable environment then uncoordinated service compositions at 5% service failures.

5.5 Ease of Use for Service Consumers

The ease of use for service consumers is improved because a selected service composition that uses the coordination architecture can offer a much higher service composition reliability (see Section 5.3) and failure resilience (see Section 5.4). A service consumer remains unaware that the coordinating architecture is used or even that a service composition is used at all.

If a service composition fails, a service consumer would have to discover another service that suits his requirements. The discovery and selection of a new service will take some time and is likely to annoy service consumers. However, the usage of the coordination architecture also introduces a delay for a service consumer. Figure 5.7 shows the minimal, maximal and mean performance times for the coordinated and

non-coordinated evaluation scenarios. The complete results for the individual clients can be found in Appendix D.



(a) min performance

(b) max performance

(c) mean performance

Figure 5.7: Minimal, maximal and mean performance

Due to the setup of the examples a complete failure can be discovered much quicker in the non-coordinated service compositions. This explains the quicker execution at a higher service failure rate as a service composition is more likely to fail straight-away. On the other hand, the coordinator will try all available service compositions, which causes the experienced delay and increases with a higher service failure rate as a fail-

over to another service compositions becomes more likely.

The mean performance time for coordinated service compositions shows an overall delay of 2-3 seconds. The overall delay can be considered minimal compared with the gained ease of use for service consumers. However, this evaluation was run in the optimal environment of only one computer. Distributing the services across several computers can introduce more delays that cannot be anticipated within this deliverable. This remains to be done in future work.

5.6 Summary

This chapter described an evaluation of our coordination architecture. The evaluation criteria included service composition reliability, failure resilience and ease of use for consumers.

The service composition reliability was evaluated for the four different service failure rates. The coordination architecture proved to be very effective on improving the service composition reliability. For 5% and 10% service failures, the coordination architecture could still guarantee a 100% reliability, falling to 99% at 20% service failures. Even at the highest service failure rate of 40% could the coordination architecture achieve a better service composition reliability than a not coordinated service composition at the lowest service failure rate of 5%.

The failure resilience was evaluated for the two different failure handling mechanisms, fail-over to another service instance and fail-over to another service composition. The first failure handling mechanism increases the failure resilience for specific service types. The achieved improvement is good for low service failures and still acceptable for the highest service failure rate of 40%.

The second failure mechanism proved to be very effective for failure resilience. As presented for service composition reliability, the coordination architecture achieves a better failure resilience even at the highest service failure rate of 40% could than a not coordinated service composition at the lowest service failure rate of 5%. However, the experienced delay for service consumers increased by about 2-3 seconds overall for using the coordination architecture within an optimised evaluation environment. The overall delay appears negligible considering the gained ease of use for service consumers.

Chapter 6

Conclusion

This deliverable described the design and implementation of a coordination architecture to support service compositions in the unstable service availability environment of the DBE. The coordination architecture was evaluated with respect to higher service composition reliability, failure resilience and ease of use for consumers.

This chapter summarises the work described in this deliverable and outlines its contributions to the state of the art. This deliverable is then concluded with a discussion of related research issues that remain open for future work.

6.1 Document Summary

Chapter 1 introduced the problem of providing reliable and failure resilient service compositions in the DBE. It motivated the challenges of dealing with unstable availability of services for designing service compositions, and the use of standard-based and open-source technologies. Based on the high-level goals for coordinating Web service compositions, their implementation into the contributions of this deliverable was stated. The contributions covered the design and implementation of the coordination architecture to coordinate BPEL service compositions and the evaluation of the coordination architecture.

Chapter 2 provided background information for the DBE and service compositions. Two service composition languages were introduced and compared. Finally, four related approaches to service compositions were detailed and evaluated based on their usage for the DBE project.

Chapter 3 we defined the requirements that were identified for the coordination architecture. The following sections described the functional model of the architecture and the architecture overview, elaborating on the design of the individual components that participate in the deployment and execution process of coordinated service compositions. Finally, the advanced failure handling mechanisms were defined.

Chapter 4 detailed the implementation of the different components that support the coordination architecture. The implementation presented in this chapter was based mainly on Java and used the ActiveBPEL engine for executing BPEL service compositions.

Finally in Chapter 5, we presented an evaluation of the coordination architecture. In separate sections, we evaluated how our architecture achieves the objectives from the requirements for service composition reliability, failure resilience and ease of use

for consumers.

6.2 Contributions

This deliverable addressed the problem of developing a coordination architecture for the DBE with the capability to support service compositions under constantly changing availability of services. The main contributions of this deliverable are the coordination architecture, introducing a coordinator and a ranking service component, advanced failure handling mechanisms and the evaluation of the coordination architecture that demonstrated the higher service composition reliability, failure resilience and ease of use for service consumers.

The coordination architecture introduced the coordinator to enable automatic fail-over to another service composition at run-time. This was achieved by deploying several service compositions that offer equivalent functionality while using different services within their composition as a set that is represented and coordinated by the coordinator. The coordinator receives a consumer's request and chooses at run-time the highest ranked service composition out of the provided set. If a selected service composition fails, the coordinator determines the next highest ranked service composition and redirects the consumer's request to this service composition. The Ranking Service was introduced within the coordination architecture as a light-weight Web service that has access to QoS information for services within the DBE, such as failure and performance history of a service that can be used to create a ranking of available and equivalent services. This functionality can be used by the coordinator to retrieve the highest ranked service out of a given set of services and to update the QoS information about a service for example after a service failure.

The coordination architecture contributed a two-step failure handling mechanism. The first failure handling mechanism provided fail-over to another service instance while the second failure handling mechanism provided fail-over to another service composition, which is provided by the already summarised coordinator component. The first mechanism took advantage of the fact that services within the DBE are identified by a unique identifier and their endpoint information is retrieved dynamically at run-time. If more than one service instance implements the same service type it reuses this identifier. Therefore, at run-time there might be several service instances available for the same service type, which enables the coordination architecture to fail-over to another service instance.

Finally, this deliverable contributed an evaluation of the coordination architecture with respect to service composition reliability, failure resilience and ease of use for consumers. The coordination architecture was run with 5%, 10%, 20% and 40% service failures and compared with individual service compositions. The result showed that the coordination architecture could absorb service failures of 5% and 10% completely while service failures of 20% were covered with a 99% guarantee. Even service failures of 40% resulted in a 91% reliability, which is a better result than for individual service compositions at 5% service failures (90%). However, the usage of the coordination architecture at a high percentage of service failures caused a deterioration of performance.

Appendix A

Client Results

A.1 Client Results for Coordinated Service Compositions

	service failures %	success	failures	client failures %	total calls
Client-1	5	500	0	0	500
Client-2	5	500	0	0	500
Client-3	5	500	0	0	500
Client-4	5	500	0	0	500
Client-5	5	500	0	0	500
Client-1	10	500	0	0	500
Client-2	10	500	0	0	500
Client-3	10	500	0	0	500
Client-4	10	500	0	0	500
Client-5	10	500	0	0	500
Client-1	20	493	7	1.4	500
Client-2	20	499	1	0.2	500
Client-3	20	496	4	0.8	500
Client-4	20	498	2	0.4	500
Client-5	20	499	1	0.2	500
Client-1	40	451	49	9.8	500
Client-2	40	457	43	8.6	500
Client-3	40	453	47	9.4	500
Client-4	40	462	38	7.6	500
Client-5	40	450	50	10	500

Table A.2: Client results for coordinator

A.2 Client Results for Booking-11 Service Composition

	service failures %	success	failure	client failures %	total calls
Client-1	5	452	48	9.6	500
Client-2	5	442	58	11.6	500
Client-3	5	457	43	8.6	500
Client-4	5	455	45	9	500
Client-5	5	458	42	8.4	500
Client-1	10	395	105	21	500
Client-2	10	415	85	17	500
Client-3	10	413	87	17.4	500
Client-4	10	414	86	17.2	500
Client-5	10	390	110	22	500
Client-1	20	329	171	34.2	500
Client-2	20	322	178	35.6	500
Client-3	20	319	181	36.2	500
Client-4	20	330	170	34	500
Client-5	20	324	176	35.2	500
Client-1	40	172	328	65.6	500
Client-2	40	181	319	63.8	500
Client-3	40	173	327	65.4	500
Client-4	40	177	323	64.6	500
Client-5	40	192	308	61.6	500

Table A.4: Client results for booking-11 service (not coordinated)

A.3 Client Results for Booking-12 Service Composition

	service failures %	success	failures	client failures %	total calls
Client-1	5	447	53	10.6	500
Client-2	5	439	61	12.2	500
Client-3	5	459	41	8.2	500
Client-4	5	449	51	10.2	500
Client-5	5	442	58	11.6	500
Client-1	10	398	102	20.4	500
Client-2	10	421	79	15.8	500
Client-3	10	404	96	19.2	500
Client-4	10	404	96	19.2	500
Client-5	10	402	98	19.6	500
Client-1	20	304	196	39.2	500
Client-2	20	311	189	37.8	500
Client-3	20	311	189	37.8	500
Client-4	20	330	170	34	500
Client-5	20	315	185	37	500
Client-1	40	149	351	70.2	500
Client-2	40	175	325	65	500
Client-3	40	182	318	63.6	500
Client-4	40	174	326	65.2	500
Client-5	40	190	310	62	500

Table A.6: Client results for booking-12 service (not coordinated)

A.4 Client Results for Booking-13 Service Composition

	service failures %	success	failures	client failures %	total calls
Client-1	5	447	53	10.6	500
Client-2	5	443	57	11.4	500
Client-3	5	449	51	10.2	500
Client-4	5	451	49	9.8	500
Client-5	5	455	45	9	500
Client-1	10	399	101	20.2	500
Client-2	10	408	92	18.4	500
Client-3	10	408	92	18.4	500
Client-4	10	402	98	19.6	500
Client-5	10	398	102	20.4	500
Client-1	20	313	187	37.4	500
Client-2	20	316	184	36.8	500
Client-3	20	326	174	34.8	500
Client-4	20	309	191	38.2	500
Client-5	20	319	181	36.2	500
Client-1	40	146	354	70.8	500
Client-2	40	179	321	64.2	500
Client-3	40	184	316	63.2	500
Client-4	40	187	313	62.6	500
Client-5	40	198	302	60.4	500

Table A.8: Client results for booking-13 service (not coordinated)

A.5 Client Results for Booking-14 Service Composition

	service failures %	success	failures	client failures %	total calls
Client-1	5	458	42	8.4	500
Client-2	5	458	42	8.4	500
Client-3	5	442	58	11.6	500
Client-4	5	449	51	10.2	500
Client-5	5	449	51	10.2	500
Client-1	10	404	96	19.2	500
Client-2	10	408	92	18.4	500
Client-3	10	405	95	19	500
Client-4	10	411	89	17.8	500
Client-5	10	419	81	16.2	500
Client-1	20	328	172	34.4	500
Client-2	20	311	189	37.8	500
Client-3	20	321	179	35.8	500
Client-4	20	342	158	31.6	500
Client-5	20	329	171	34.2	500
Client-1	40	185	315	63	500
Client-2	40	184	316	63.2	500
Client-3	40	211	289	57.8	500
Client-4	40	176	324	64.8	500
Client-5	40	184	316	63.2	500

Table A.10: Client results for booking-14 service (not coordinated)

A.6 Client Results for Booking-15 Service Composition

Client-5	10	398	102	20.4	500
	service failures %	success	failures	client failures %	total calls
Client-1	5	451	49	9.8	500
Client-2	5	452	48	9.6	500
Client-3	5	442	58	11.6	500
Client-4	5	445	55	11	500
Client-5	5	445	55	11	500
Client-1	10	391	109	21.8	500
Client-2	10	414	86	17.2	500
Client-3	10	414	86	17.2	500
Client-4	10	406	94	18.8	500
Client-1	20	321	179	35.8	500
Client-2	20	309	191	38.2	500
Client-3	20	336	164	32.8	500
Client-4	20	324	176	35.2	500
Client-5	20	297	203	40.6	500
Client-1	40	171	329	65.8	500
Client-2	40	179	321	64.2	500
Client-3	40	197	303	60.6	500
Client-4	40	174	326	65.2	500
Client-5	40	161	339	67.8	500

Table A.12: Client results for booking-15 service (not coordinated)

Appendix B

Failure Rates

B.1 Difference Between Set and Real Service Failures at 5%

	set failure-rate %	failures	total calls	real failure-rate %
email-1	5	68	1335	5.09
email-2	5	69	1297	5.32
hotel-1	5	63	1055	5.97
hotel-2	5	33	536	6.16
hotel-3	5	55	1060	5.19
flight-1	5	59	1108	5.32
flight-2	5	57	1118	5.1
flight-3	5	27	568	4.75

Table B.1: Service usage and real failure rate for 5%

B.2 Difference Between Set and Real Service Failures at 10%

	set failure-rate %	failures	total calls	real failure-rate %
email-1	10	139	1378	10.09
email-2	10	146	1383	10.56
hotel-1	10	128	1135	11.28
hotel-2	10	69	547	12.61
hotel-3	10	104	1119	9.29
flight-1	10	118	1220	9.67
flight-2	10	132	1249	10.57
flight-3	10	64	646	9.91

Table B.2: Service usage and real failure rate for 10%

B.3 Difference Between Set and Real Service Failures at 20%

	set failure-rate %	failures	total calls	real failure-rate %
email-1	20	291	1537	18.93
email-2	20	305	1461	20.88
hotel-1	20	269	1234	21.8
hotel-2	20	126	636	19.81
hotel-3	20	238	1258	18.92
flight-1	20	306	1511	20.25
flight-2	20	292	1567	18.63
flight-3	20	175	823	21.26

Table B.3: Service usage and real failure rate for 20%

B.4 Difference Between Set and Real Service Failures at 40%

	set failure-rate %	failures	total calls	real failure-rate %
email-1	40	625	1597	39.14
email-2	40	619	1631	37.95
hotel-1	40	586	1512	38.76
hotel-2	40	328	806	40.96
hotel-3	40	580	1525	38.03
flight-1	40	960	2496	38.46
flight-2	40	990	2501	39.58
flight-3	40	486	1282	37.91

Table B.4: Service usage and real failure rate for 40%

Appendix C

Ranking Results

C.1 Usage of Ranking Service

	make Ranking calls	min calls	additional calls
5% service failure	2794	2500	294
10% service failure	3115	2500	615
20% service failure	3886	2500	1386
40% service failure	6052	2500	3552

Table C.1: Ranking service usage

C.2 Usage of Coordinated Service Compositions

	set failure-rate %	failures	total calls	real failure-rate %
booking-11	5	35	548	6.39
booking-12	5	27	568	4.75
booking-13	5	30	572	5.24
booking-14	5	24	560	4.29
booking-15	5	27	546	4.95
booking-11	10	53	608	8.72
booking-12	10	64	646	9.91
booking-13	10	62	642	9.66
booking-14	10	65	612	10.62
booking-15	10	70	607	11.53
booking-11	20	137	706	19.41
booking-12	20	175	823	21.26
booking-13	20	154	819	18.8
booking-14	20	169	805	20.99
booking-15	20	138	748	18.45
booking-11	40	500	1230	40.65
booking-12	40	486	1282	37.91
booking-13	40	535	1317	40.62
booking-14	40	460	1266	36.33
booking-15	40	455	1184	38.43

Table C.2: Usage of coordinated BPEL service compositions

Appendix D

Performance Results

D.1 Performance in Clients for Coordinated Service Composition

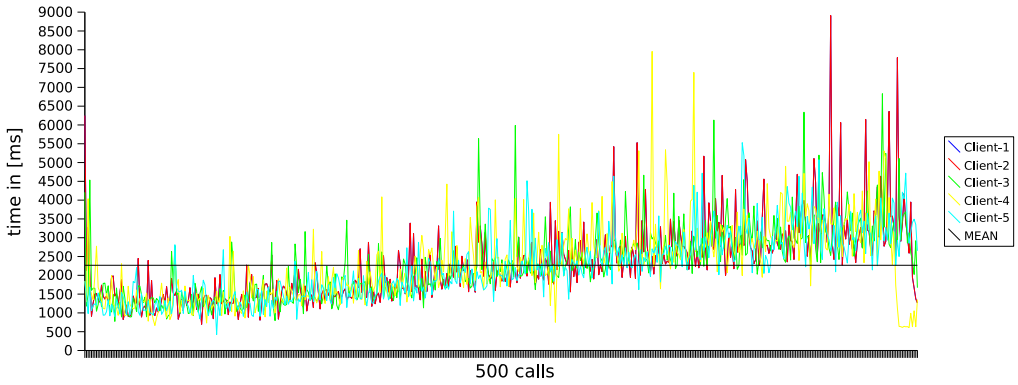


Figure D.1: Clients using coordination architecture 5% service failure

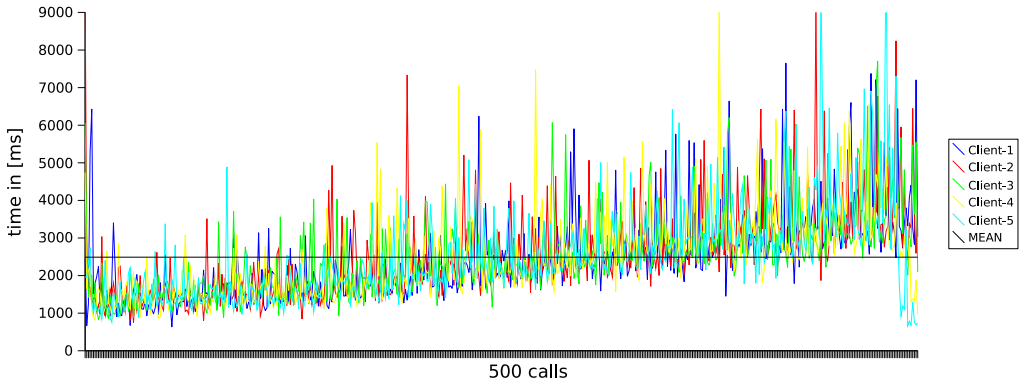


Figure D.2: Clients using coordination architecture 10% service failure

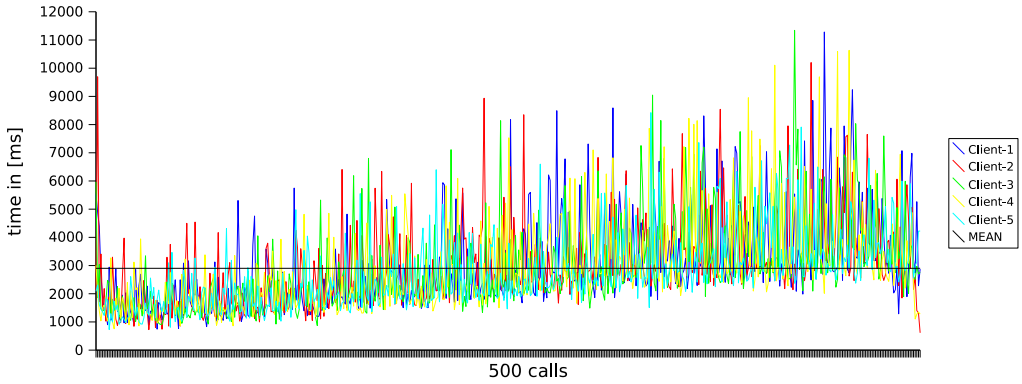


Figure D.3: Clients using coordination architecture 20% service failure

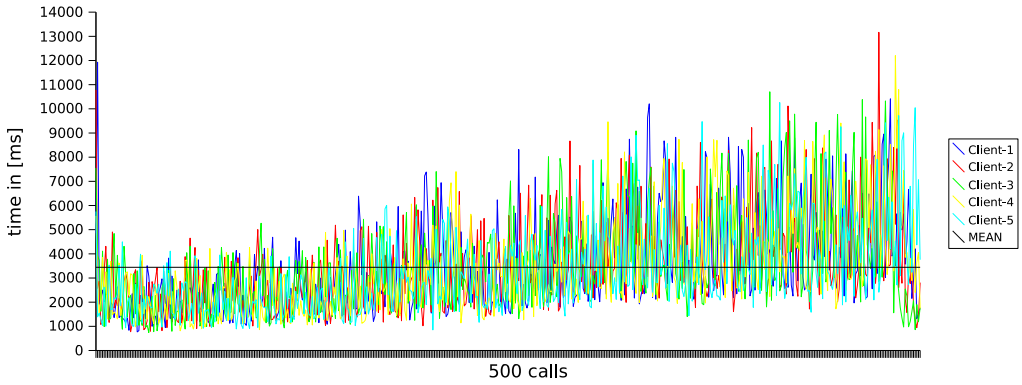


Figure D.4: Clients using coordination architecture 40% service failure

D.2 Performance in Clients for Booking-11

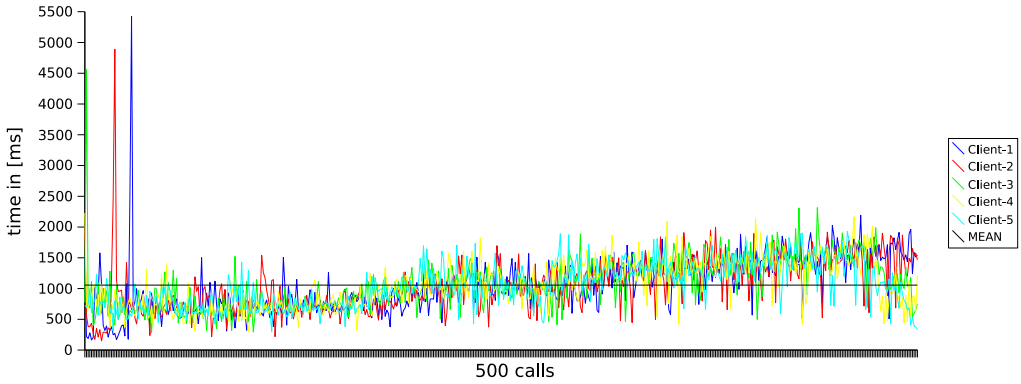


Figure D.5: Clients using Booking-11 5% service failure

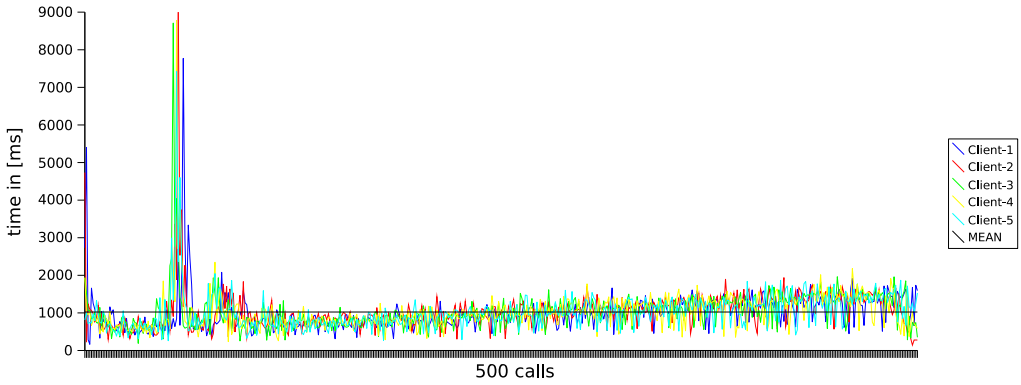


Figure D.6: Clients using Booking-11 10% service failure

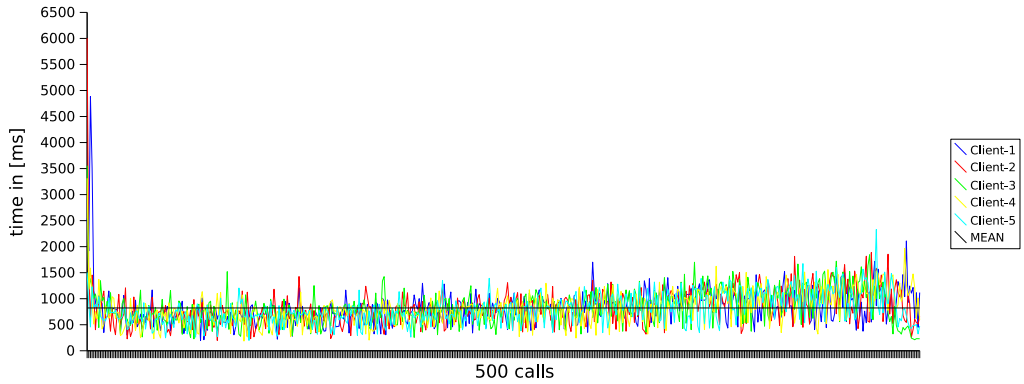


Figure D.7: Clients using Booking-11 20% service failure

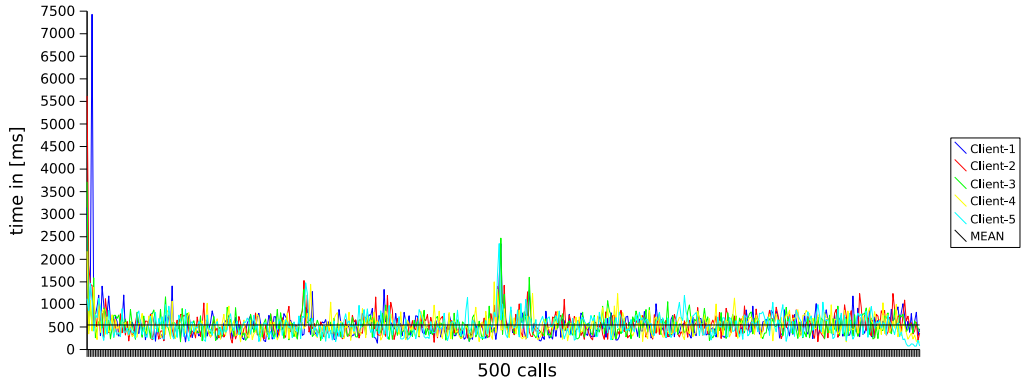


Figure D.8: Clients using Booking-11 40% service failure

D.3 Performance in Clients for Booking-12

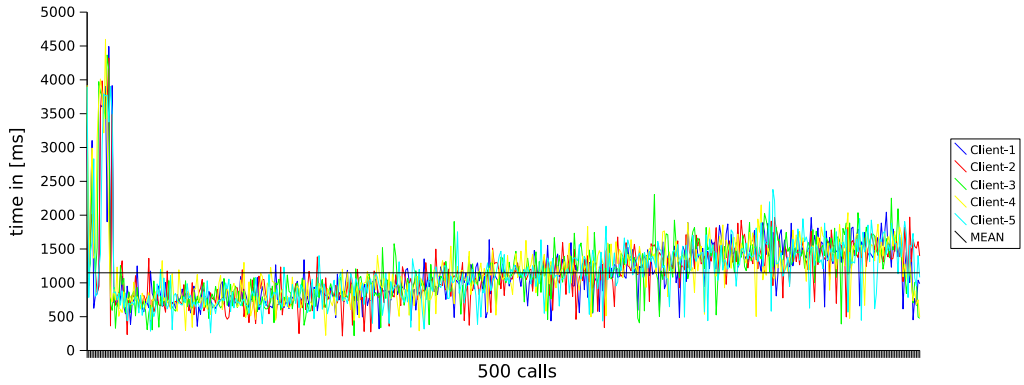


Figure D.9: Clients using Booking-12 5% service failure

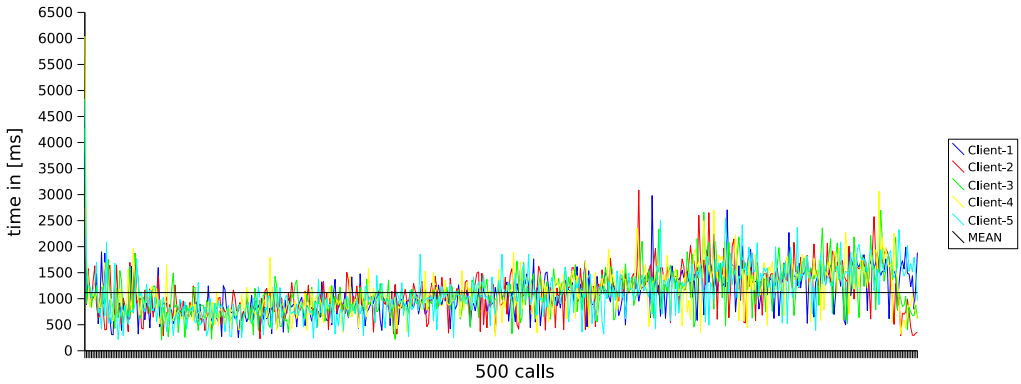


Figure D.10: Clients using Booking-12 10% service failure

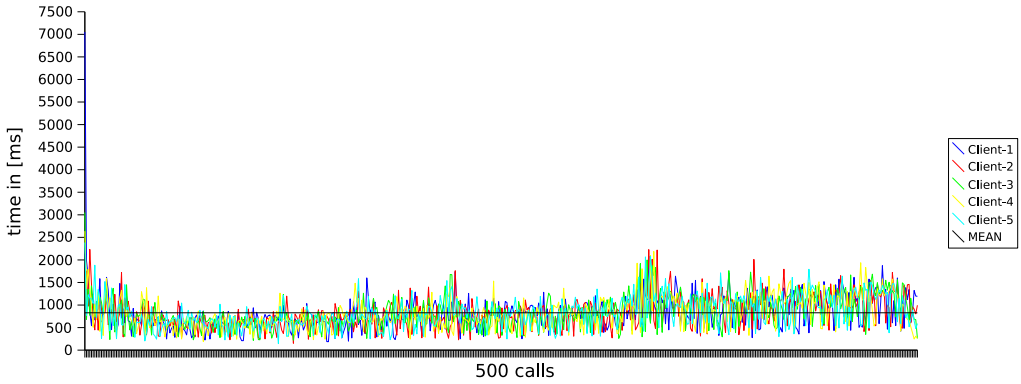


Figure D.11: Clients using Booking-12 20% service failure

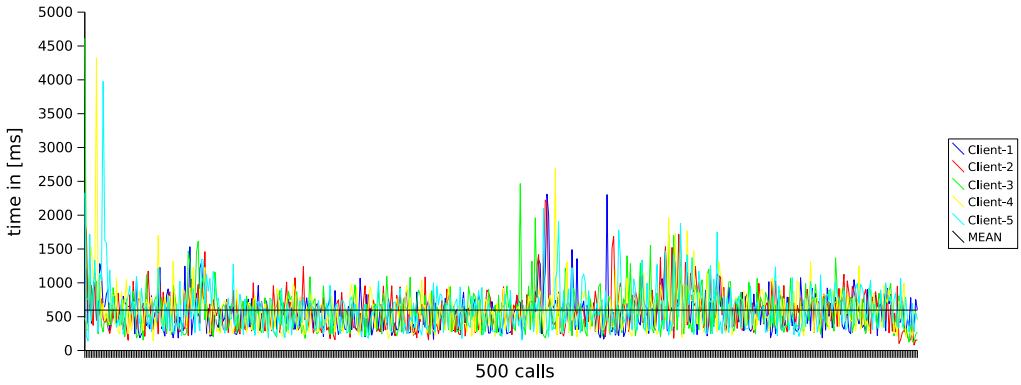


Figure D.12: Clients using Booking-12 40% service failure

D.4 Performance in Clients for Booking-13

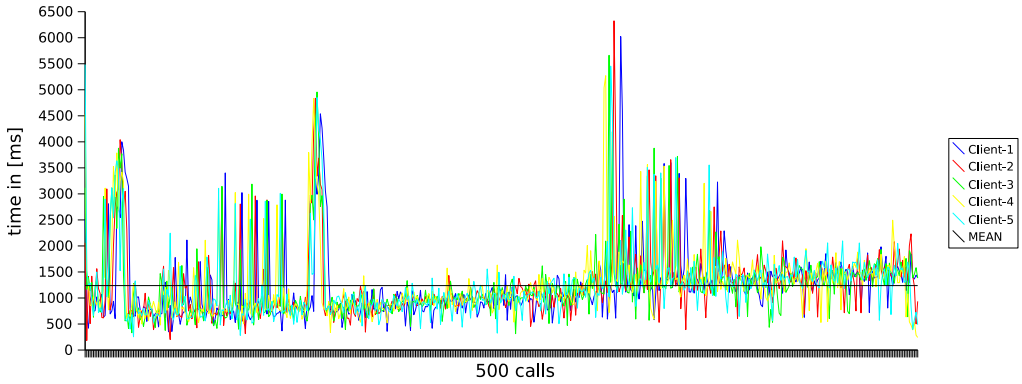


Figure D.13: Clients using Booking-13 5% service failure

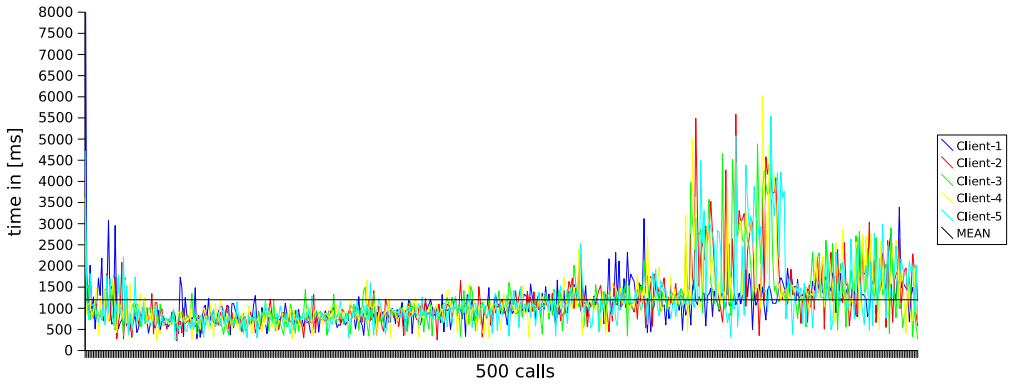


Figure D.14: Clients using Booking-13 10% service failure

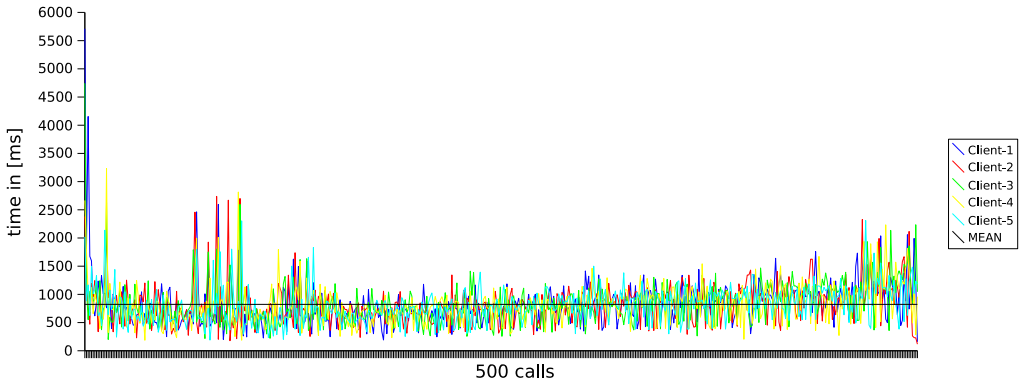


Figure D.15: Clients using Booking-13 20% service failure

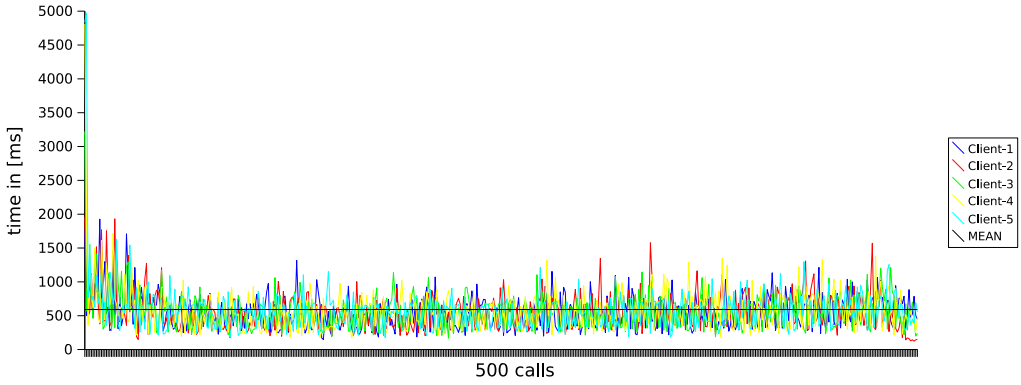


Figure D.16: Clients using Booking-13 40% service failure

D.5 Performance in Clients for Booking-14

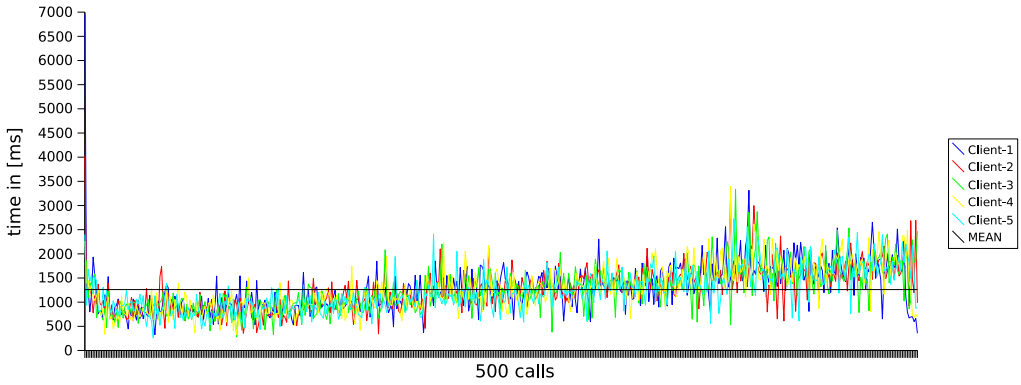


Figure D.17: Clients using Booking-14 5% service failure

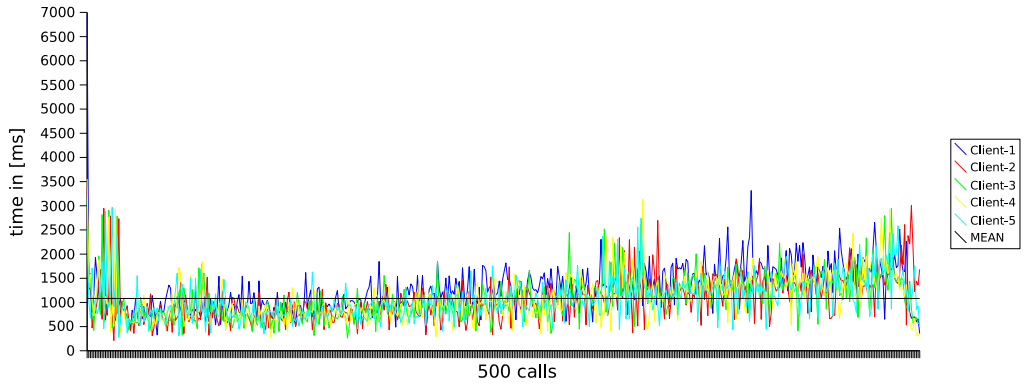


Figure D.18: Clients using Booking-14 10% service failure

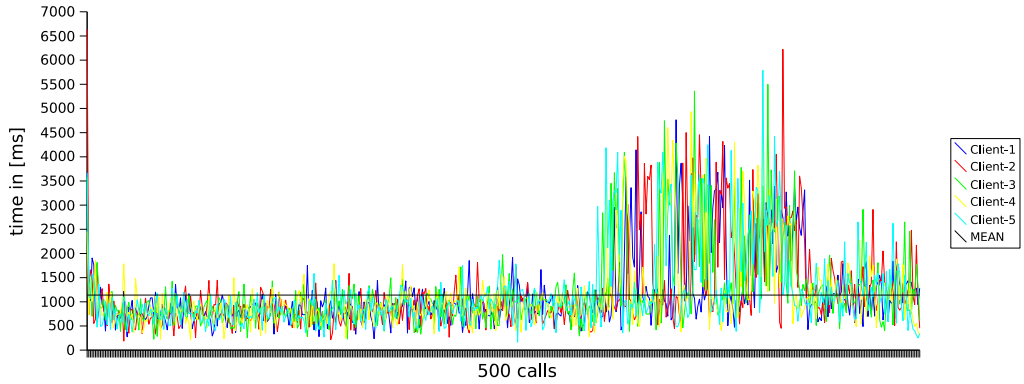


Figure D.19: Clients using Booking-14 20% service failure

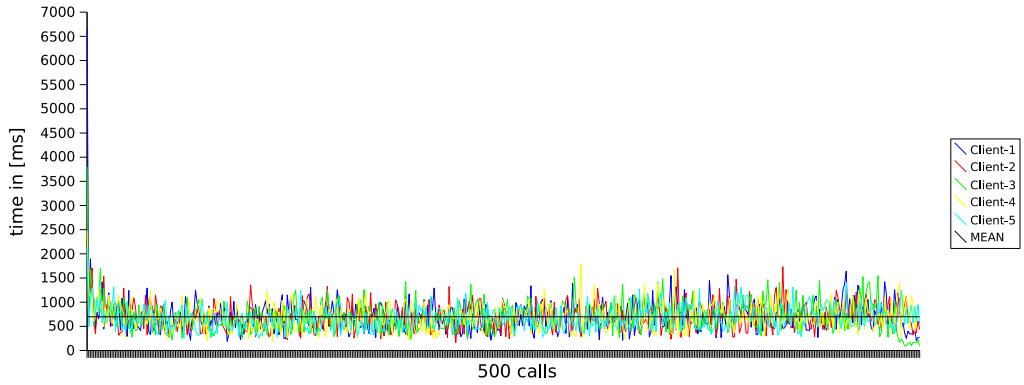


Figure D.20: Clients using Booking-14 40% service failure

D.6 Performance in Clients for Booking-15

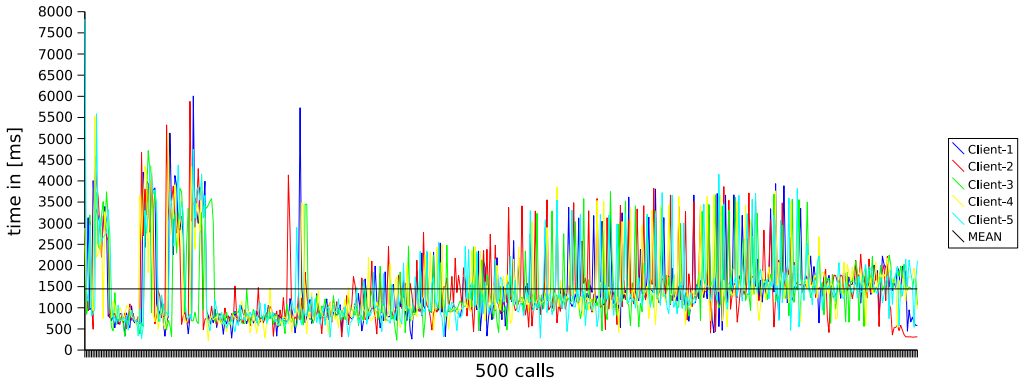


Figure D.21: Clients using Booking-15 5% service failure

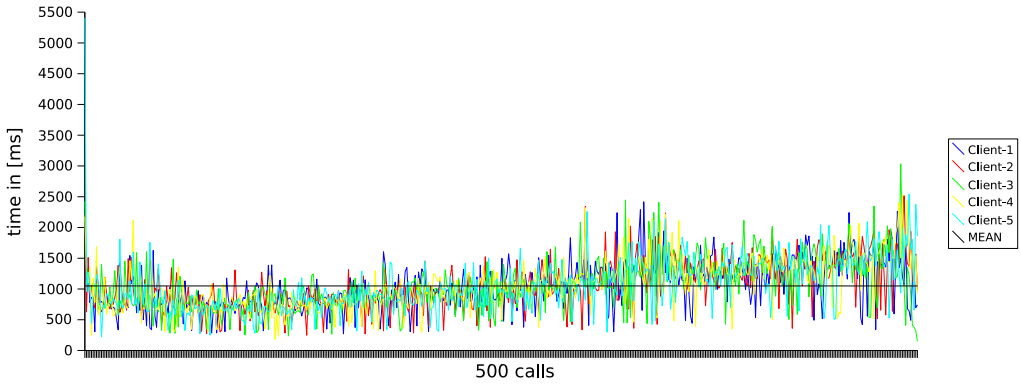


Figure D.22: Clients using Booking-15 10% service failure

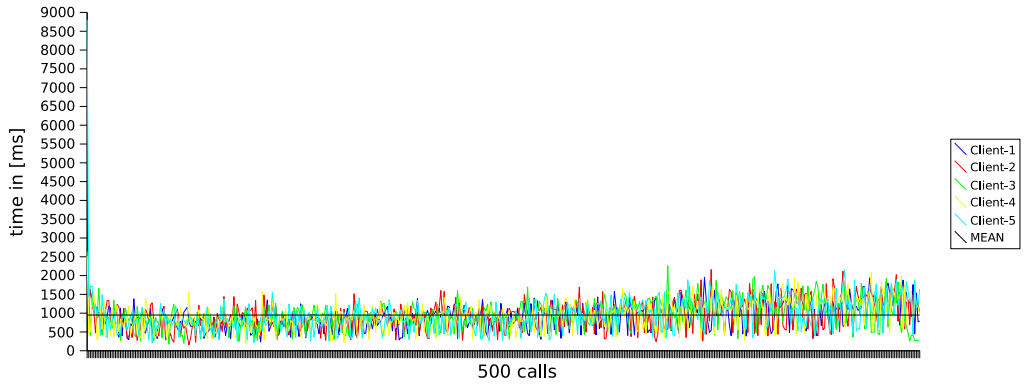


Figure D.23: Clients using Booking-15 20% service failure

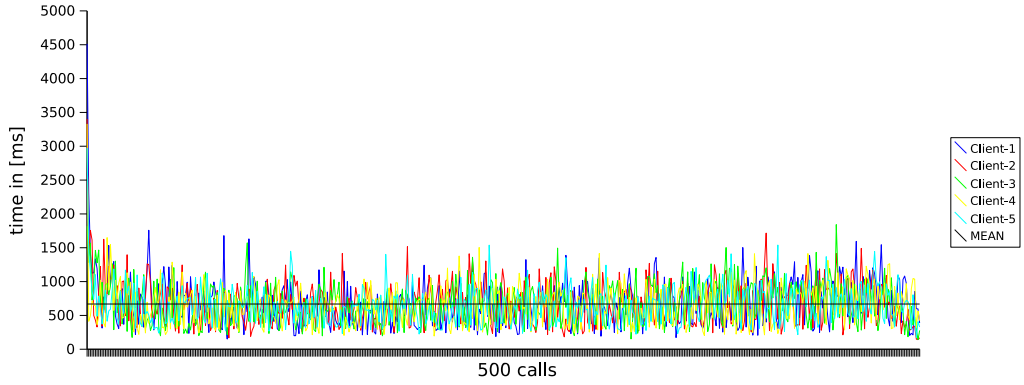


Figure D.24: Clients using Booking-15 40% service failure

Bibliography

[1] CVS - Concurrent Versions System. URL http://www.nongnu.org/cvs/.

[2] ActiveBPEL LLC. ActiveBPEL Engine, Open Source BPEL Server, 2004-2006. URL http://www.activebpel.org.

[3] Vikas Agarwal, Girish Chafle, Koustuv Dasgupta, Neeran Karnik, Arun Kumar, Sumit Mittal, and Biplav Srivastava. Synthy: A System for End to End Composition of Web Services. *Journal of Web Semantics*, 3(4), September 2005.

[4] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint Driven Web Service Composition in METEOR-S. In *Services Computing, 2004 IEEE International Conference on (SCC'04)*, pages 23–30. IEEE Computer Society, September 2004.

[5] BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services. Version 1.1, May 2003. URL http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[6] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, January/February 2003.

[7] Jorge Cardoso, John A. Miller, Jianwen Su, and Jeff Pollock. Academic and Industrial Research: Do Their Approaches Differ in Adding Semantics to Web Services? volume 3387 of *LNCS*, pages 14–21, July 2005.

[8] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL). Version 1.1, March 2001. URL http://www.w3.org/TR/wsdl.

[9] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, November 2004. URL http://www.daml.org/services/owl-s/1.1/owl-s.pdf.

[10] Francisco Cubera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next Step in Web Services. *Communications of the ACM*, Vol. 46 (10):29–34, October 2003.

[11] Francisco Curbera, Mattew J. Duftler, Rania Khalaf, William A. Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, March/April 2002.

[12] Francisco Curbera, Matthew J. Duftler, Rania Khalaf, Nirmal Mukhi, William A. Nagy, and Sanjiva Weerawarana. BPWS4J. Published online by IBM at http://www.alphaworks.ibm.com/tech/bpws4j, 2002.

[13] Francisco Curbera, Frank Leymann, Dieter Roller, Marc-Thomas Schmidt, Matthias Kloppmann, Frank Skrzypczak, and Francis Parr. Web Services Flow Language (WSFL). Version 1.0, May 2001. URL `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[14] James Curran and Robert Blackburn. Panacea or White Elephant? A Critical Examination of the Proposed New Small Business Service and Response to the DTI Consultancy Paper. *Regional Studies*, 34(2):181–206, 2000.

[15] Dominik Dahlem, David McKitterick, Lotte Nickel, Jim Dowling, Bartek Biskupski, and Rene Meier. Binding- and Port-Agnostic Service Composition using a P2P SOA. In *International ICSOC Workshop on Dynamic Web Processes (DWP)*, December 2005.

[16] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language. W3C Recommendation, February 2004. URL `http://www.w3.org/TR/owl-ref/`.

[17] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[18] Digital Business Ecosystem. Digital Business Ecosystem. http://www.digital-ecosystem.org/, 2004-2006. URL `http://www.digital-ecosystem.org/`.

[19] Digital Business Ecosystem. DBE Studio. http://dbestudio.sourceforge.net/, October 2005-2006. URL `http://dbestudio.sourceforge.net/`.

[20] Digital Business Ecosystem. P2P Client and Server (ServENT). http://sourceforge.net/projects/swallow, 2005-2006. URL `http://sourceforge.net/projects/swallow`.

[21] Digital Business Ecosystem. Project Swallow. http://swallow.sourceforge.net/, 2005-2006. URL `http://swallow.sourceforge.net/`.

[22] Richard Fikes, Pat Hayes, and Ian Horrocks. DAML Query Language, April 2003. URL `http://www.daml.org/2003/04/dql/`.

[23] Richard Fikes, Jessica Jenkins, and Gleb Frank. JTP: A System Architecture and Component Library for Hybrid Reasoning. In *7th World Multiconference on Systemics, Cybernetics, and Informatics*, July 2003.

[24] The Apache Software Foundation. Apache Maven Project. URL `http://maven.apache.org/`.

[25] The Apache Software Foundation. Log4j. Version 1.2.8, April 2004. URL `http://logging.apache.org/log4j/docs/index.html`.

[26] The Apache Software Foundation. Apache ANT Project, June 2005. URL `http://ant.apache.org/`.

[27] The Apache Software Foundation. Web Services - Axis. Version 1.2.1, June 2005. URL http://ws.apache.org/axis/.

[28] The Eclipse Foundation. The Eclipse Open Extensible IDE. Version 3.1, 2005. URL http://www.eclipse.org.

[29] Nektarios Gioldasis, Nikos Pappas, Fotis Kazasis, George Anestis, and Stavos Christodoulakis. A P2P and SOA infrastructure for distributed ontology-based knowledge management. 2004.

[30] Object Management Group. Common Object Request Broker Architecture: Core Specification. OMG Specification 3.0.3, March 2004. URL http://www.omg.org/docs/formal/04-03-01.pdf.

[31] The HSQLDB Development Group. HSQLDB. Version 1.8.0, April 2006. URL http://www.hsqldb.org/.

[32] Thomas Heistracher, Thomas Kurz, Claudius Masuch, Pierfranco Ferronato, Miguel Vidal, Angelo Corallo, Gerard Briscoe, and Paolo Dini. Pervasive Service Architecture for a Digital Business Ecosystem. In *First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT)*, June 2004.

[33] Business Process Management Initiative. Business Process Modelling Language (BPML), June 2005. URL http://www.bpmi.org/.

[34] Benchaphon Limthanmaphon and Yanchun Zhang. Web Service Composition Transaction Management. In *Fifteenth Australian Database Conference (ADC)*, volume 27, pages 171 – 179. ACM Conferences in Research and Practice in Information Technology, January 2004.

[35] LSDIS Lab, University of Georgia. METEOR-S: Semantic Web Services and Processes. URL http://lsdis.cs.uga.edu/projects/meteor-s/.

[36] Daniel J. Mandell and Sheila A. McIlraith. A Bottom-Up Approach to Automating Web Service Discovery, Customization and Semantic Translation. In *International Semantic Web Conference*, volume 2870 of *LNCS*, pages 227–241, October 2003.

[37] Umardand Shripad Manikrao and T.V.Prabhakar. Dynamic Selection of Web Services with Recommendation System. In *International Conference on Next Generation Web Services Practices*, pages 117–121, August 2005.

[38] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, March/April 2001.

[39] Microsoft, IBM, Sun Microsystems Inc., BEA Systems, and SAP AG. Web Services Addressing (WS-Addressing). W3C Member Submission, August 2004. URL http://www.w3.org/Submission/ws-addressing/.

[40] Nikola Milanovic and Miroslaw Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51 – 59, November/December 2004.

[41] Richard Monson-Haefel. *J2EE Web Services*. Addison-Wesley, 2004.

[42] OASIS Committee. Universal Description, Discovery and Integration (UDDI). Published Specification Version 2, July 2002. URL `http://www.uddi.org`.

[43] Nicole Oldham, Christopher Thomas, Amit P. Sheth, and Kunal Verma. METEOR-S Web Service Annotation Framework with Machine Learning Classification. In *SWSWPC*, volume 3387 of *LNCS*, pages 137–146, July 2004.

[44] Michael P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of Fourth Conference on Web Information Systems Engineering (WISE)*, pages 3 – 12. IEEE, December 2003.

[45] Chris Peltz. Web Services Orchestration and Choreography. *IEEE Computer Science*, 36(10):46–52, October 2003.

[46] Preeda Rajasekaran, John Miller, Kunal Verma, and Amith Sheth. Enhancing Web Services Description and Discovery to Facilitate Orchestration. In *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, July 2004.

[47] Eric S. Raymond. The Cathedraal and the Bazaar, 1998. URL `http://firstmonday.dk/issues/issue3_3/raymond`.

[48] Jan Sacha and Jim Dowling. A gradient topology for master-slave replication in peer-to-peer environments. *Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2005.

[49] Naveen Srinivasan, Massimo Paolucci, and Katia P. Sycara. An Efficient Algorithm for OWL-S Based Semantic Search in UDDI. In *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, volume 3387 of *LNCS*, pages 96–110, July 2004.

[50] Biplav Srivastava and Jana Koehler. Web Service Composition - Current Solutions and Open Problems. In *Proceedings of Workshop on Planning for Web Services (ICAPS)*, June 2003.

[51] Michael Stal. Web Services: beyond component-based computing. *Communications of the ACM*, 45(10):71–76, October 2002.

[52] SUN. Fetish Advanced Directory Architecture (FADA). http://fada.sourceforge.net/, 2001-2005. URL `http://fada.sourceforge.net/`.

[53] Satish Thatte and Microsoft. Web Services for Business Process Design, 2001. URL `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

[54] Chang tseh Hsieh and Binshan Lin. Internet commerce for small businesses. *Industrial Management and Data Systems*, 98(3):pages 113–119, 1998.

[55] Wil van der Aalst. Don't Go with the Flow: Web Services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, January/February 2003.

[56] Kunal Verma, Karthik Gomadam, Amit Sheth, John Miller, and Zixin Wu. The METEOR-S Approach for Configuring and Executing Dynamic Web Processes. In *Third International Conference on Service Oriented Computing (ICSOC)*, December 2005.

[57] Kunal Verma, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar, and John Miller. METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Information Technology and Management*, 6(1):17–39, 2005. ISSN 1385-951X.

[58] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[59] Steve Vinoski. Where is Middleware? *IEEE Internet Computing*, 6(2):83–85, March/April 2002.

[60] Steve Vinoski. Integration with Web Services. *IEEE Internet Computing*, 7(6): 75–77, November/December 2003.

[61] Werner Vogels. Web Services Are Not Distributed Objects. *IEEE Internet Computing*, 7(6):59–66, November/December 2003.

[62] W3C. XML Path Language. Version 1.0, November 1999. URL `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[63] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *ER*, volume 2813 of *LNCS*, pages 200–215, October 2003.