University of St Andrews

School of Computer Science

Data transfer optimization in orchestrated Web service workflows

Metodi Ewgeniew Metodiew

Matriculation Number: 100017632

Dissertation

MSc in Advanced Software Engineering

29 August 2011

Acknowledgments

I would like to express my gratitude to my thesis supervisor Dr. Adam Barker for introducing me to the field of Web services. His guidance and expertise were crucial to the success of this project.

I would like to thank all people at the University of St Andrews for making the last year a truly memorable experience.

I would also like to thank my family who has supported me during the course of this dissertation.

Finally, I would like to thank my friends Kosjo and Elka for making me believe that it is never too late to go back to school.

Abstract

Web services are the basic building blocks on which is formed the distributed computing on the Internet. Stand alone services provide remarkable functionality and computing resources; however, some of the most sophisticated applications on the Web are comprised of many separate services, coordinated to work together in order to accomplish a common goal. Often, the coordination of these Web services is achieved through the use of Web service orchestration. With the orchestration model, a central node – the workflow engine, manages the interaction between the Web services – all messages pass through the workflow engine. Workflow orchestration is characterized by simplified error handling and high robustness. However, with the increase of the number of participating Web services and the amount of transferred data, the centralized orchestration model begins to suffer from lack of scalability. The solution, proposed in this thesis, is to increase the performance by optimizing the data transfer. This is achieved by introducing a new participant in the workflow – Workflow Speedup Proxy (WSProxy). WSProxy is designed to keep the large transfers local to the Web services by exchanging references instead of the actual data. As a consequence of localizing the transfers, the network traffic is minimized and the memory and computational requirements on the workflow engine are significantly decreased.

Declaration

I hereby certify that this dissertation, which is approximately 10500 words in length, has been composed by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree. This project was conducted by me from June/2011 to August/2011 towards fulfillment of the requirements of the University of St Andrews for the degree of MSc under the supervision of Dr. Adam Barker.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Date: ………………………….. Signature: ………………………….……………

Metodi Ewgeniew Metodiew

List of Figures

Figure 1. Choreography 1
Figure 2. Orchestration 2
Figure 3. Orchestration with WSProxy 3
Figure 4. High-level architecture 14
Figure 5. Message flow 15
Figure 6. WSProxy architecture 16
Figure 7. Model-View-Controller 17
Figure 8. Main classes in Web Services 18
Figure 9. Proxy cache 19
Figure 10. Message processing 20
Figure 11. Main database classes 24
Figure 12. Distance measurement classes 25
Figure 13. Geographical positioning of the instances 27
Figure 14. Sequential orchestration 29
Figure 15. Sequential orchestration with Proxy in US 29
Figure 16. Sequential with proxy in Ireland 29
Figure 17. Sequential workflow in Taverna [7] 29
Figure 18. Results sequential scenario with 1KB 30
Figure 19. Results sequential scenario with 100KB 30
Figure 20. Results sequential scenario with 1MB 30
Figure 21. Fan-in orchestration 31
Figure 22. Fan-in orchestration with Proxy in US 31
Figure 23. Fan-in orchestration with Proxy in Ireland 31
Figure 24. Fan-in workflow in Taverna [7] 31
Figure 25. Results fan-in scenario with 1KB 32
Figure 26. Results fan-in scenario with 100KB 32
Figure 27. Results fan-in scenario with 1MB 32
Figure 28. Fan-out orchestration 33
Figure 29. Fan-out orchestration with proxy in US 33
Figure 30. Fan-out orchestration with proxy in Ireland 33
Figure 31. Fan-out workflow in Taverna [7] 33
Figure 32. Results fan-out scenario with 1KB 34
Figure 33. Results fan-out scenario with 100KB 34
Figure 34. Results fan-out scenario with 1MB 34

List of Listings

Listing 1 Example XML 8
Listing 2. Example XML Namespace 8
Listing 3. Example XML Schema 9
Listing 4. Validated XML Document 9
Listing 5. Example WSDL definition 11
Listing 6. Example SOAP response 12
Listing 7. Example SQL query 13
Listing 8. Message containing reference 21
Listing 9. Message containing actual data 21
Listing 10. Example of Programmable API code 22
Listing 11. WSProxy database schema 23
Listing 12. Reference report query 24

Table of Contents

Acknowledgments i
Abstract ii
Declaration iii
1. Introduction 1
  1.1. Web Services and workflows 1
  1.2. Problem statement 1
  1.3. Existing solutions 2
  1.4. The proposed solution 3
  1.5. Contribution of the project 4
  1.6. Organization of the next chapters 4
2. Related Work 5
  2.1. Orchestrating Data-Centric Workflows 5
  2.2. Eliminating the middleman 5
  2.3. The Circulate architecture 5
  2.4. Proxies to Accelerate Cloud Applications 5
  2.5. Decentralized Orchestration 5
  2.6. Service Invocation Triggers 6
  2.7. Composite Web Services 6
  2.8. Hybrid Orchestration 6
  2.9. Reference Passing 7
  2.10. Taverna Data Proxy 7
3. Background 8
  3.1. XML 8
  3.2. XML Namespaces 8
  3.3. XML Schema 9
  3.4. WSDL 9
  3.5. SOAP 12
  3.6. Java Server Pages 12
  3.7. Apache Tomcat 13
  3.8. SQL 13
4. Architecture 14
  4.1. High-level architecture 14

4.2. Message flow 15
4.3. WSProxy components 16
5. Implementation 17
5.1. Administration component 17
5.2. Web Services 18
5.3. Programmable API 22
5.4. Storage 23
5.5. Distance measurement 25
5.6. Quality assurance 26
6. Experiments 27
6.1. Introduction 27
6.1.1. Amazon EC2 instances 27
6.1.2. Regions 27
6.1.3. Workflow engine 27
6.1.4. Test Web Services 28
6.1.5. Taverna workflow elements 28
6.1.6. Measurements 28
6.2. Scenarios 28
6.2.1. Sequential scenario 29
6.2.2. Fan-in scenario 31
6.2.3. Fan-out scenario 33
6.3. Performance evaluation 35
6.3.1. Evaluation of workflows implemented in Java and Taverna 35
6.3.2. Evaluation of proxy scenarios 35
7. Future Work 37
7.1. Improvements in functionality 37
7.2. Research and Experiments 37
8. Conclusions 38
Appendix A: Installation 39
Projects 39
Build instructions 39
Security settings 40
Deployment 40
Appendix B: WSProxy Web interface 41

Main page..41
Application settings..41
Storage management..42
Message log...42
Web Services 43
Action, Configure 43
Connected Proxies 44
References...45

1. Introduction

1.1. Web Services and workflows

Web services [1] are the basic building blocks on which is formed the distributed computing on the Internet. XML Web services are the main platform for application integration, based on open standards and collaboration among people and applications. The main idea in Service Oriented Architecture (SOA) is the development of distributed applications form independent components, located on geographically distributed locations [2]. Key technologies that form the SOA architecture are the XML based languages such as SOAP and Web Services Description Language (WSDL). These technologies have been designed to support inter-component communication and information exchange in a platform independent manner, resulting in applications that are loosely coupled and not bound to a particular language or development platform.

Stand alone Web services may provide remarkable functionality or resources; however, the most sophisticated SOA applications are comprised of many separate services, coordinated to work together in order to accomplish a common goal. This coordination of Web services is often accomplished through the use of workflow technologies. Workflow is defined as the sequence of steps or operations during which documents, information or tasks are exchanged between participants, according to a predefined set of rules with the aim of achieving specific goal [3]. Implementing workflows can be achieved through two main architectural approaches: service choreography and service orchestration.

1.2. Problem statement

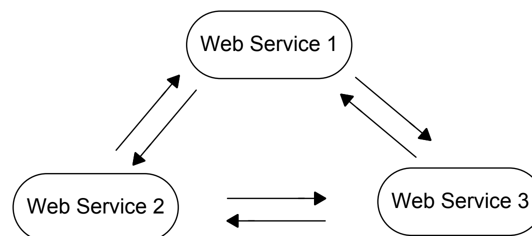


Figure 1. Choreography

In Web service choreography, the participating services exchange messages in peer-to-peer fashion and each participant is aware of its involvement in the workflow execution. This model is characterized by high throughput, scalability and low response time. However, choreography also brings increased implementation complexity, complicated error handling and recovery. Moreover, deadlocks and inefficient use of Web service resources is possible if incorrect design is applied. On the figure is shown the data flow in the choreography model, where each of the Web services is aware of the other participants.

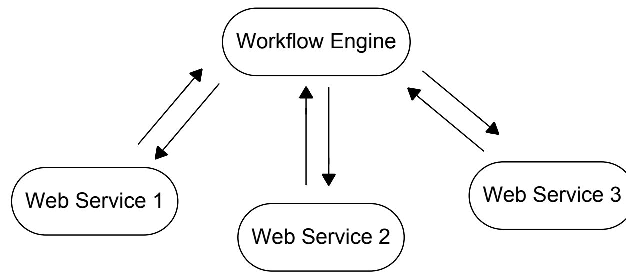


Figure 2. Orchestration

With the orchestration model of interaction, a central node – the workflow engine, manages both the control flow and the data flow. The Web services participating in orchestration workflows are not aware of their involvement in a higher level application which means that there is lower component coupling and less implementation complexity of the individual Web service. However, with the increase of the number of participating services and the amount of transferred data, the centralized orchestration model begins to suffer from lack of scalability. Transit data is transferred to the central node - wasting bandwidth, causing network bottlenecks and memory failures to the execution of the workflow.

The problem with orchestration is even more exacerbated by the fact that mobile devices, with relatively limited resources, such as smart phones and tablets are gaining more and more popularity. When such devices are used to execute workflows in an orchestrated model, they may not posses enough resources to accommodate the intermediate data. Furthermore, the devices may be subject to intermittent or disrupted connectivity which could render large data transfers impossible. On the figure is shown the data flow in the orchestrated model, where all data transits through a central node – Workflow Engine.

1.3. Existing solutions

Currently, there are several existing solutions that offer different ways to improve the performance of a centralized orchestration. Service Invocation Triggers [4] is an approach in which the workflow logic is partitioned and distributed among many different middleware components called “triggers”. These components collect the input data needed for a single Web service call and when all inputs are in place they invoke (fire) the service operation. Shortcoming of this approach is the difficult monitoring of the progress of the service invocations. There is no explicit notification in cases of execution failures and the only indication of an error is when timeout occurs.

More robust, hybrid approach that deals with the workflow performance bottleneck is the Circulate architecture [5]. Circulate is based on centralized control flow and distributed data flow. Circulate is based on lightweight middleware, a proxy, which provides a gateway and a standard API for Web service invocation. In Circulate, however, the data have to be specifically managed through the use of the API calls to invoke, upload and deliver operations. This means that the workflow should include service calls that are not directly related to the “business” logic of the process, making the composition of workflows more difficult to implement and maintain.

An implementation of a proxy which provides the functionality of passing references instead of the actual data, is Taverna Data Proxy [6]. Its main purpose is to solve the problem of Taverna application crashing because of the large volumes of data returned by the services in data-centric workflows cause Out of Memory exception. However, limitation to the data proxy is that it does not address the issues of garbage

collection of the references that are no longer needed. Furthermore, the WSDL descriptions of the services are not modified to include the new schema for the referenced types. This means that if complex types other than string are referenced by the data proxy, the WSDLs should be edited manually in order for the proxy to be used by engines different than Taverna.

1.4. The proposed solution

The solution, presented in this thesis, is to introduce a new node to the orchestrated workflow – Workflow Speedup Proxy (WSProxy). This node will modify the message traffic in such way that only small messages containing pointers to data will be returned to the workflow engine. The actual data will stay on this new node and will be forwarded to the participating Web services. The large data objects are transferred between the Web services and the central node as pointers to data, in a fashion similar to reference passing in languages such as C++.

The proposed solution retains the benefits of simplified error handling and robustness of the centralized orchestration while at the same time provides improved performance by keeping the data transfers local to the consumer Web services. Large amount of data generated from producer Web services will be stored on the proxy and then forwarded directly the consumer services at the next stages of the workflow.

WSProxy can be employed within any existing workflow engine such as Taverna [7] or it could be used by workflows implemented in high-level languages such as Java or C#. It is based on existing industry standards and tools (XML, SOAP, WSDL and Apache Tomcat) that are widely accepted and have significant user base and application support.

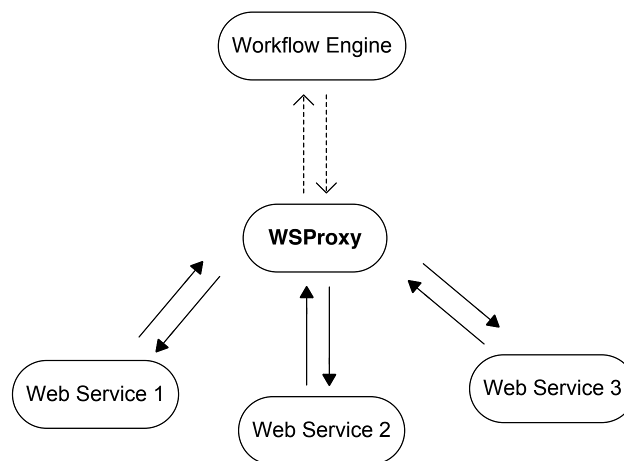


Figure 3. Orchestration with WSProxy

On the figure above, the dashed line represent control flow messages containing references to data. The solid lines represent the larger messages that contain the actual data. Most of the traffic stays between the WSProxy and the Web services.

1.5. Contribution of the project

- Workflow Speedup Proxy (WSProxy) - Web service based middleware acting as a referencing proxy to the workflow orchestration engine.
- Web based interface, with restricted access, for configuring and management of the deployed proxies.
- Web service geographic locator functionality for more informed Web service registration.
- Programmable API for access to WSProxy storage.
- Development of producer and consumer Web services used for performance evaluation.
- Sample workflows implemented in both Java and Taverna engine.
- Evaluation of different deployment scenarios on Amazon Elastic Compute Cloud (EC2).
- Examination of the architecture and implementation some possible future extensions and improvements.

1.6. Organization of the next chapters

- Chapter 2 explores the related work in the field.
- Chapter 3 introduces some basic terminology and key technologies.
- Chapter 4 describes the architecture of the solution.
- Chapter 5 explains the implementation details of the software.
- Chapter 6 is the experimental and evaluation part of the thesis.
- Chapter 7 points out some direction for future work and possible extensions.
- Chapter 8 concludes the thesis.

2. Related Work

2.1. Orchestrating Data-Centric Workflows

“Orchestrating Data-Centric Workflows” [8] is a paper that presents light-weight hybrid architecture for executing large-scale scientific workflows [9]. It retains the robustness of centralized error handling and control, but facilitates choreography by allowing services to directly exchange data. It presents a standard compliant non-disruptive solution to the problem of transferring huge amounts of scientific data. In order to realize centralized control flow and distributed data flow, the paper presents the light weight non-intrusive proxy architecture. In this architecture, the proxies need to be installed as close as possible to the participating in the workflow services. The proxies are exposed and managed through an API described by standard WSDL interface, allowing them to be built into higher level workflows.

2.2. Eliminating the middleman

Paper that investigates the problem of transferring large amounts of scientific data is “Eliminating The Middleman: Peer-to-Peer Dataflow” [10]. It also uses the non-intrusive proxy architecture presented above. In the paper, a series of experiments common to many scientific applications, sequential, fan-in, fan-out, are performed. The analysis of the results shows that significant improvements can be achieved. An average performance benefit of 2.03 to 2.83 times in the worst-case remote scenario, and a 3.47 to 3.88 times in the best case scenario is observed.

2.3. The Circulate architecture

Another paper that presents the hybrid solution is “The Circulate architecture: avoiding workflow bottlenecks caused by centralized orchestration” [5]. It focuses on Web Service based implementation of orchestration model of central control, but a choreography model for optimized and distributed data transfer. The performance analysis done in the paper concludes that substantial reduction in overhead results in 2-4 fold performance benefit. An End-to-End pattern through the Montage workflow demonstrates that by combination of patterns - 8 fold performance benefits is possible.

2.4. Proxies to Accelerate Cloud Applications

A similar solution that addresses the problem of bottlenecks caused by the client-server interaction in the current cloud computing is presented in the paper “Using Proxies to Accelerate Cloud Applications” [11]. In the paper, the authors show the potential for accelerating data transfer by exploiting the proxy architecture. A model in which several proxies may take different data-centric roles: proxy-cloud; proxy-proxy, data caching inside a proxy and intermediate proxy processing is investigated. In this model, the proxies operate on behalf of the application in order to accelerate its performance. The authors have conducted an extensive evaluation of their proxy network using nodes deployed across the Planet Lab test bed and have observed a significant improvement.

2.5. Decentralized Orchestration

Very often scientific workflows are modeled as Direct Acyclic Graphs (DAG). When orchestrating service-oriented DAGs, the workflow engine becomes a performance bottleneck, resulting in increased execution time. A paper that proposes a decentralized orchestration solution to the problem is “Decentralized

Orchestration of Service-Oriented Scientific Workflows" [12]. The paper introduces architecture for splicing and deploying DAGs onto peer-to-peer proxy network. In this solution, the initial DAG is divided into set of vertices and distributed across a group of proxies. In this model, each proxy executes in parallel a part of the original DAG. The paper proposes that each proxy will be deployed at a network location corresponding to certain conditions like, network distance, security, policy, etc. The main idea of the approach is that by breaking the original workflow and distributing it among multiple proxies, the performance bottleneck associated with centralized workflow processing will be eliminated. A major advantage of the solution is that it is non-intrusive in its nature. The participating Web services are not needed to be modified or redeployed. Moreover, existing workflow scripts can be translated by into DAG based workflow and automatically deployed across the proxy network. The architecture of the proposed solution consists of generic proxies that can concurrently execute any workflow definition. This is achieved by passing the part of the DAG that is to be processed by a proxy is passes as an executable object. The proxies invoke Web services and store any intermediate data locally. Because the proxies are organized in peer-to-peer network, when they need to exchange data, they can pass data directly to each other. This ability for direct transfer of the intermediate data eliminates the performance bottleneck of the centralized orchestration. In order to evaluate the performance benefits, various experiments are performed in the paper. The results show that improvement from 30% to 150% in the various test scenarios is achieved.

2.6. Service Invocation Triggers

Another approach to solving the performance degradation of data-centric workflows, by employing decentralized execution, is explored in the paper "Service Invocation Triggers: A Lightweight Routing Infrastructure for Decentralized Workflow Orchestration" [4]. The paper introduces the notion of invocation triggers, which act as proxies for individual service invocations. These triggers serve as a buffer - collecting the required input data before they invoke the service. In addition, the triggers forward service outputs to exactly those destinations where they are actually needed – in this way effectively supporting multicast invocations. The key part in implementing this functionality is the decomposition of the workflows into sequential parts. Once the trigger is of a Web service receives all required data, it commences the execution and passes its outputs to the subsequent service triggers. As a result, no data is passed to a central orchestration engine – the workflow is executed in fully decentralized way. However this is a limiting solution and it prevents the use of proxy technology; moreover it requires the workflow to be changed before enactment.

2.7. Composite Web Services

In the paper "Decentralized Orchestration of Composite Web Services" [12] is introduced an architecture for routing the messages directly from the producing Web service to the consuming one. It explores the ramifications of decomposition of a workflow scripts written in BPEL4WS and executing the different parts in parallel. The authors make observation that decentralized orchestration brings increased complexity both in error handling and error recovery. Furthermore, decentralized execution is prone to incorrect design and decomposition, which might lead to potential deadlocks or non-optimal usage of system resources. The paper investigates these and other build and runtime issues; however it does not investigate the problem of how to optimize the DAG workflows or how to deploy the workflow across the proxy network.

2.8. Hybrid Orchestration

Extensive analysis and experiments of a hybrid model for coordinating and executing workflows is performed in the master thesis "Hybrid Web Service Orchestration" [13]. The model is based on the proxy architecture. The proxies are working together with the workflow engine, but attempt to be a non-disruptive extension to the traditional orchestration model. In "Hybrid Web Service Orchestration"

project, series of experiments configured and deployed on the PlanetLab [14] scientific network are evaluated. The paper notes that the placement of the proxies is crucial to the performance of the workflows. If the proxies are deployed in locally to the workflow engine, there is significant degradation in performance and even the traditional orchestration model performs better. On the other hand, it is observed that when the workflow engine is moved further away from the enacted Web services, the performance benefits increase substantially.

2.9. Reference Passing

Solution to the bottleneck in orchestrated workflows is investigated in the paper "Towards Reference Passing in Web Service and Workflow-based Applications" [15]. The paper explores the idea of passing references instead of the actual data. The authors introduce a new type of BPEL variable – so called reference variable. With this variable, only pointers to the large data objects are exchanged between the Web services and the orchestrating workflows. In their solution the pointers are managed externally to the workflow system. The architecture of the solution consists of Reference Resolution System (RRS), where the central component is the RRS itself. The RSS connects both Web service interfaces and holds a set of adapters that ensure the interoperability between the different data sources. Each adapter consists of a lookup service and an execution service. The lookup service is used to store queries and information. The execution service is dedicated to executing the queries on the data source where the actual referenced data are located. In this paper, the authors have not addressed the issues of visibility and scope of the reference variables. Another issue that is not investigated is the matter of deleting the referenced data after it is no longer needed, e.g. a sort of garbage collection mechanisms.

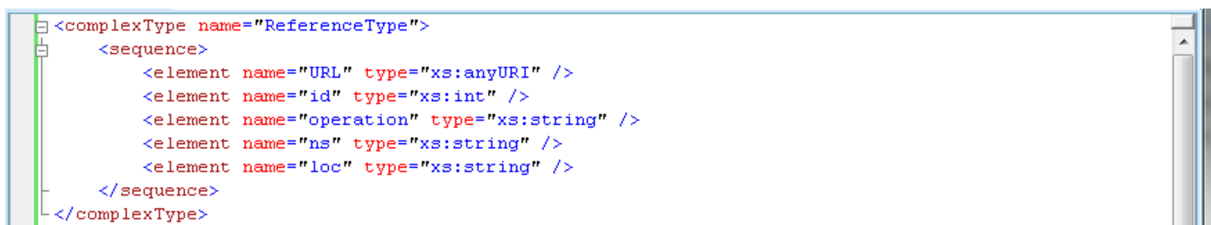
2.10. Taverna Data Proxy

An implementation, specific to Taverna workflow engine, which provides the functionality of passing references instead of the actual data, is Taverna Data Proxy [6]. Its main purpose is to solve the problem of Taverna application crashing because of the large volumes of data returned by the services in data-centric workflows cause Out of Memory exception. Web services are added to the proxy by specifying their WSDL interface together with some description. In order for a Web service to return reference, it requires manually selecting the output or return value that is to be referenced. Once an element is selected for referencing, instead of the actual value a URL to a file is returned. If a proxy encounters a reference in the input parameters of an operation, it automatically dereferences the data that is contained in the corresponding proxy. However, limitation to the data proxy is that it does not address the issues of garbage collection of the references that are no longer needed. Furthermore, the WSDL descriptions of the services are not modified to include the new schema for the referenced types. This means that if complex types other than string are referenced by the data proxy, the WSDLs should be edited manually in order for the proxy to be used by engines different than Taverna.

3. Background

3.1. XML

Extensible Markup Language (XML)[1], is a markup language used to transport and store data. It consists of a set of rules for composing documents that are suitable for machine processing. Although its design is focused on documents, one of its most common uses is for representation of data structures in Web services. XML provides the building blocks for many other languages, such as XML Schema, Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL). There are three major terms used to describe parts of an XML document: tags, elements, and attributes. Below is given a sample excerpt of document that illustrates the terms:



```
<complexType name="ReferenceType">
    <sequence>
        <element name="URL" type="xs:anyURI" />
        <element name="id" type="xs:int" />
        <element name="operation" type="xs:string" />
        <element name="ns" type="xs:string" />
        <element name="loc" type="xs:string" />
    </sequence>
</complexType>
```

Listing 1 Example XML

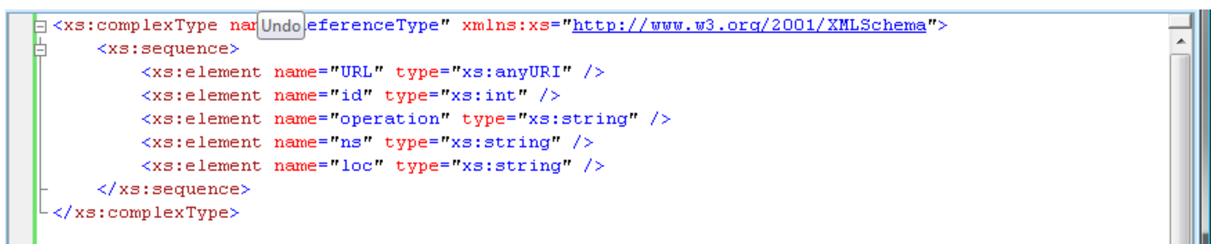
- A tag is the text between the left angle bracket (<) and the right angle bracket (>). There are starting tags (such as <sequence>) and ending tags (such as </sequence>).

- An XML element is the starting tag, the ending tag, and everything in between. In the sample above, the <complexType … > element contains one <sequence> element and five <element … /> elements.

- An attribute is a name-value pair inside the starting tag of an element. In this example, name and type are attributes of the <element name="**URL**" type="**xs:anyURI**" /> element.

3.2. XML Namespaces

XML's power lies in its flexibility, the fact that it provides functionality to developers to define their own tags to describe data. However, there is a problem with this flexibility - how to distinguish between the many custom defined tags with the same name. In the previous example, the document includes tag <sequence>, which in this case, is used to describe the sequence of members in a complex type. In another document, for recording system events for example, the <sequence> tag can be used to describe the sequence of events in a system. As a consequence, to be able to differentiate between these tags, XML documents use namespace attributes. To make tags unique, developers define namespace, and then include the namespace in front of the tag's name:



```
<xs:complexType nar Undo eferenceType" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:sequence>
        <xs:element name="URL" type="xs:anyURI" />
        <xs:element name="id" type="xs:int" />
        <xs:element name="operation" type="xs:string" />
        <xs:element name="ns" type="xs:string" />
        <xs:element name="loc" type="xs:string" />
    </xs:sequence>
</xs:complexType>
```

Listing 2. Example XML Namespace

Now, the xmlns attribute introduces the xs namespace prefix, in that way the <xs:sequence>, along with all other prefixed tags, is qualified and in combination with its namespace is considered to be unique.

Here, it must be mentioned that the string in the namespace definition "**http://www.w3.org/2001/XMLSchema**" is just a string. It has the appearance of an URL, it even point to a document located on the Internet, but actually it is not an URL. The only requirement to this string is to be unique. The widely accepted convention is for these strings to be represented as URLs.

3.3. XML Schema

XML schema describes the structure of other XML documents [16]. The purpose of a schema is to provide functionality for defining how valid XML should be formed. It works in a way that is similar to the way that database schema works - describes the shape of the data and the elements it should contain. XML schema is an XML document, which means that it can be processed just like any other document. In addition to the datatypes defined in the XML schema specification, developers can also create their own. In order to promote reuse, all new types can be derived from other already defined types. In the example below, a schema for Reference type element that reuses the previously introduced complex type ReferenceType is defined.

```
<xs:schema elementFormDefault="qualified"
           targetNamespace="http://www.proxy_accelerator.net/Schemas/Reference/v1"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="ReferenceType">
        <xs:sequence>
            <xs:element name="URL" type="xs:anyURI"/>
            <xs:element name="id" type="xs:int"/>
            <xs:element name="operation" type="xs:string"/>
            <xs:element name="ns" type="xs:string"/>
            <xs:element name="loc" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ReferenceTypeWrap">
        <xs:sequence>
            <xs:element name="ref" type="tns:ReferenceType"
                xmlns:tns="http://www.proxy_accelerator.net/Schemas/Reference/v1"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Reference">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ref" type="tns:ReferenceType"
                    xmlns:tns="http://www.proxy_accelerator.net/Schemas/Reference/v1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Listing 3. Example XML Schema

The schema can be used to validate the structure and content of the following element:

```
<ns3:Reference xmlns:ns3="http://www.proxy_accelerator.net/Schemas/Reference/v1" >
    <ns3:ref>
        <ns3:URL>http://ec2-17-201-58-184.compute-1.amazonaws.com:8084</ns3:URL>
        <ns3:id>10</ns3:id>
        <ns3:operation>putData</ns3:operation>
        <ns3:ns>unused</ns3:ns>
        <ns3:loc>unused</ns3:loc>
    </ns3:ref>
</ns3:Reference>
```

Listing 4. Validated XML Document

3.4. WSDL

Web Services Description Language (WSDL) is a XML based language used to describe Web services as a collection of endpoints operating on messages in either document or procedural style [17]. The operations and messages are described abstractly and then bound to concrete protocol and format.

WSDL documents contain machine-readable description of how a Web service should be invoked, what operations are exposed, what types of parameter it expects and what are the return types. WSDL is often used to define Web services in SOAP transported over HTTP protocol. The structure of WSDL document files is composed of six high-level elements: types, messages, port types, bindings, ports and service. All these elements are enclosed in a top level element named definitions.

- **Definitions** – This is the root WSDL element. This element is the place where all global namespaces are declared. Its targetNamespace attribute is used to set the default namespace for the WSDL. All elements used without namespace prefix will be associated with this namespace.

- **Types** – In this element are contained all types that participate in the exchange of messages. Each type is declared either inline or referenced. The referencing of external schemas is through the use of import or include directives.

- **Messages** – This element consists of one or more parts. Each part is associated with a type. This association depends on the WSDL style and is realized through messages-typing attribute. Depending on the binding style, the typing attribute is element or type.

- **Port types** – Port type is a collection of abstract operations and messages that serves the role of an operation interface of the Web service. Each operation in the collection has a combination of input and output elements. The order of these elements defines the message exchange pattern of the operation. There are four patterns: one-way; request-response; solicit-response; and notification.

- **Bindings** - Bindings describe the concrete implementation details of a specific port type. It has two attributes: name and type. The name attribute is used to identify the binding elsewhere in the WSDL document. The type attribute is a reference to the name of the port type that is being bound. For SOAP bindings, an extensibility element `<soap:binding … >` is used. Inside this element, the style of service “document” or “rpc” along with the transport protocol, usually HTTP, is specified.

- **Services** – Defines a collection of ports that expose bindings. For each port, there is an associated name attribute and inside the port is specified the actual location of the Web service that fulfills the WSDL contract.

The latest version of the WSDL is 2.0, however this version has not gained industry popularity, and therefore the WSDL examined and implemented by this thesis is version 1.1.

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="http://producer/"
             xmlns:cmn="http://common_types/test/services/v1"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://producer/" name="ProducerService">
    <types>
        <xs:schema targetNamespace="http://producer/">
            <xs:import namespace="http://common_types/test/services/v1"
                       schemaLocation="http://ec2.amazonaws.com:8080/TestServices/ProducerDocService?xsd=1"/>
            <xs:complexType name="getDataType">
                <xs:sequence>
                    <xs:element name="size" type="xs:int" minOccurs="0" />
                </xs:sequence>
            </xs:complexType>
            <xs:element name="getData" type="tns:getDataType" />
        </xs:schema>
    </types>
    <message name="getData">
        <part name="parameters" element="tns:getData" />
    </message>
    <message name="getDataResponse">
        <part name="parameters" element="cmn:putData" />
    </message>
    <portType name="ProviderDocService">
        <operation name="getData">
            <input message="tns:getData" />
            <output message="tns:getDataResponse" />
        </operation>
    </portType>
    <binding name="ProducerServicePortBinding" type="tns:ProducerDocService">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
        <operation name="getData">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="ProducerDocService">
        <port name="ProducerServicePort" binding="tns:ProviderServicePortBinding">
            <soap:address location="http://ec2.amazonaws.com:8080/TestServices/ProviderDocService" />
        </port>
    </service>
</definitions>
```

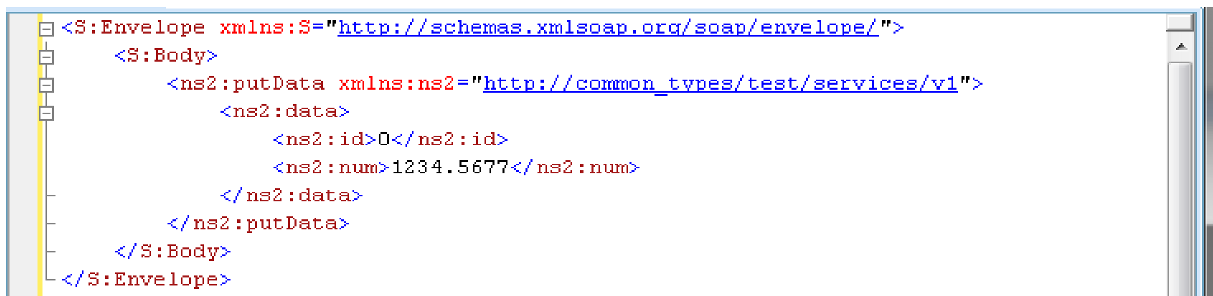
Listing 5. Example WSDL definition

On the listing above, is given WSDL definition of a simple Web service that exposes one operation for producing test data in form of an array of complex types. In the definitions elements, the name of the description is set to **ProducerService** and the target namespace is set to **http://producer/**. In addition, four more namespaces are introduced, one of which is the default namespace for all elements: xmlns="**http://schemas.xmlsoap.org/wsdl/**".

In the types section, a top level XML schema element is declared. Next, with element <xs:import …> a reference to external schema is introduced. With this reference, all types that are declared in the external schema are now visible inside the WSDL. Besides the imported schema, one inline type - **getDataType** and one element of this type - **getData** are declared. In the message sections, the types previously declared and imported in types section are used to define the message parts. The first message **getData** has part of type **getDataType**. The second message has part of type **cmn:putData** which is defined in the external schema. In the port type section, the operation **getData** is defined along with its message pattern of request-response messages. In the binding section, HTTP transport type is specified along with document binding style. The binding style of the input and output message is literal. The service section links the port and the binding and defines the endpoint address to the service to be: **http://ec2.amazonaws.com:8080/TestServices/ProviderDocService**

3.5. SOAP

SOAP [18] is widely accepted communication protocol, intended for information exchange in decentralized and distributed environments. The fundamental role of SOAP is to define the specifications and XML formats for Web service's message transfers. It consists of well formatted XML fragments enclosed in SOAP elements. SOAP is used to describe the application interface data in XML and to specify how to implement the invocation of the services' Remote Procedure Call (RPC). SOAP is specifically designed to overcome the limitations of DCOM and CORBA which, because of the network firewalls and other restrictions, have difficulties communicating over the Internet. The most important quality of SOAP is that is has been widely accepted by the industry and it has been implemented across great variety of software and hardware platforms.



Listing 6. Example SOAP response

On listing above, is given example of a message response to a request made to a producer Web service. The top XML element is <S:Envelope …> inside which is declared the namespace for all other SOAP elements: xmlns:S=http://schemas.xmlsoap.org/soap/envelope/. Inside the <S:Body> element, are enclosed the elements of the message payload: <ns2:putData …>. Message payload elements have their own XML namespace: xmlns:ns2="http://common_types/test/services/v1".

3.6. Java Server Pages

Java Server Pages (JSP) is a technology that provides developers with means to create dynamic content. It was designed in response to the critiques that Java does not offer enough support for effective Web development. JSP pages are compiled into servlets and executed by a servlet container.

The main features of JSP are:

- Language for developing Web pages that describe how to process request and compose the response.
- Language constructs that provide access to server-side functionality.
- Libraries that provide functionality for extending the JSP language.

JSP is often used to implement the visualization part in a Model-View-Controller architecture style application.

3.7. Apache Tomcat

Apache Tomcat is an open source implementation of the Java Servlet and JSP technologies [19]. Tomcat has four major components dedicated to the execution of specific tasks:

- Catalina – This is the servlet container. It acts as a host to servlets (objects).
- Coyote – HTTP connector component that listens on a dedicated TCP port. All received requests are passed to the Tomcat engine.
- Jasper – JSP engine. Parses the JSP pages and then compiles them to servlets. The servlets are in turn passed for execution to Catalina.
- Jasper2 – Improvement over Jasper. Add some new features such as tag library pooling, background compilation, recompilation on-the-fly, etc.

3.8. SQL

Structured Query Language (SQL) [20] is a declarative language that has become the standard for data manipulation and creation in relational databases. SQL has become popular in Web applications due to its capabilities to create complex storage sub-systems. Some of the major commands in SQL that provide create, read, update and delete (CRUD) functionality are:

- CREATE – Used to creation new objects.
- SELECT – Retrieving of information
- UPDATE – Modifying existing information.
- DELETE – Removing existing information

```
1 CREATE TABLE IF NOT EXISTS PARAMETER_METADATA (
2         ID INTEGER PRIMARY KEY AUTOINCREMENT,
3         WEB_SERVICE_ID INTEGER NOT NULL,
4         INTERFACE TEXT NOT NULL,
5         [OPERATION] TEXT NOT NULL,
6         [PARAMETER] TEXT NOT NULL,
7         REF_TYPE TEXT NOT NULL,
8         FOREIGN KEY(WEB_SERVICE_ID) REFERENCES WEB_SERVICES(ID)
9 )
```

Listing 7. Example SQL query

On the listing above, is shown an example of a SQL query for creation of table PARAMETER_DATA with a foreign key reference to another table WEB_SERVICES.

4. Architecture

4.1. High-level architecture

WSProxy is developed as a middleware application that is positioned between the Workflow engine and the Web services. It is implemented and deployed as a servlet accessible to its clients as a SOAP Web service. The main responsibility of WSProxy is to accept, examine, in some cases modify, and forward the SOAP messages that come from the Workflow engine or the Web Services.

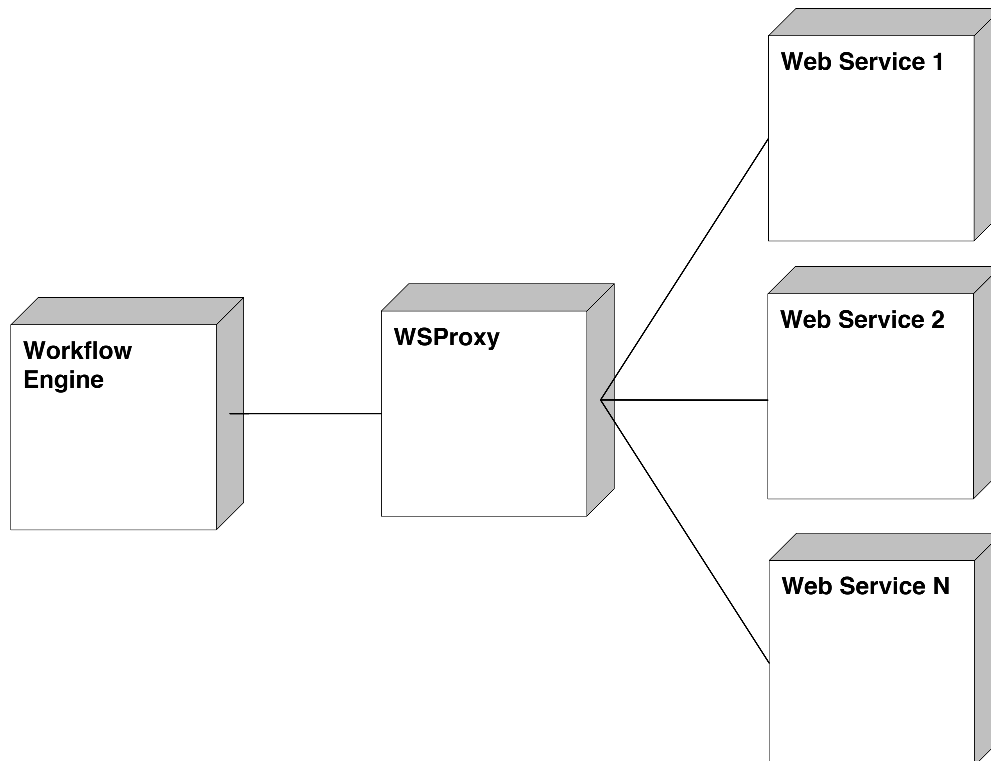


Figure 4. High-level architecture

On the figure above, is shown Web service orchestration configuration with **WSProxy**, placed between the **Workflow Engine** and the enacted **Web Services**. WSProxy acts as a repository for large objects that can be exchanged in an orchestrated workflow. It modifies the message traffic is such way, that the Workflow engines receive small self-contained references instead of the actual data.

WSProxy is a concurrent application and can process many Workflow requests in parallel. Single proxy is capable of serving an entire workflow with many participating Web services. However, by design WSProxy does not limit the workflows to only one proxy. In cases with certain network and security constraints in place, it is possible for multi-proxy configurations to be arranged.

In WSProxy is also implemented distance measurement functionality that can assist administrators in choosing the optimal proxy-Web Services configuration. Best performance gains are achieved when the network distance between the WSProxy and Web Services is as minimal as possible.

The whole application is contained in a single WAR file and is deployed in a servlet container such as Apache Tomcat or Oracle GlassFish Server. [21].

4.2. Message flow

The main objective of WSProxy is to filter and store the large data objects contained in the payloads of the message traffic. To achieve that goal, WSProxy examines the SOAP messages and replaces the large XML data elements with small self-contained references.

WSProxy receives requests from the Workflow engine, examines them and if there are references among the parameters replaces the references with the actual data. On the other hand, when WSProxy receives responses from the Web services, if the responses are marked for referencing, it replaces the actual data with a reference.

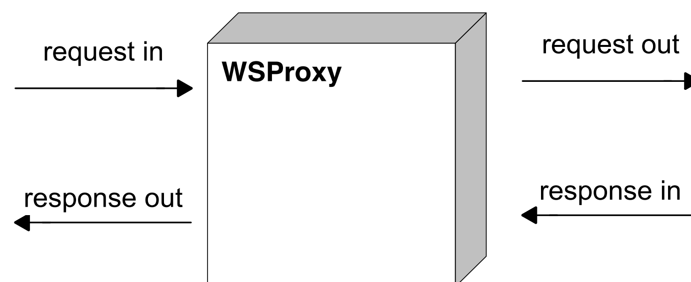


Figure 5. Message flow

In a single invocation of a Web Service made by the Workflow engine, there are four distinct messages:

- **Request In** – This is the original message sent from the Workflow engine. It is possible that some of the parameters in the message are references. If such parameters are encountered, they are resolved to the actual data they point, before leaving the proxy.

- **Request Out** – This is the modified workflow engine request that is leaving the proxy. All references are replaced with the actual data they point. Because there are no references in the outgoing message, the participating Web services remain unaware of the WSProxy existence.

- **Response In** – This is the original message that is returned by the Web Service in response to Workflow engine request. It is possible that the return parameter of the response is marked for referencing. If so, the data is extracted from the response payload and stored on the proxy.

- **Response Out** – This is the Web Service response that is being forwarded to the Workflow engine. The large data has been filtered from the messages and replaced with a reference. In this way, the traffic to the Workflow engine is kept to a minimum.

4.3. WSProxy components

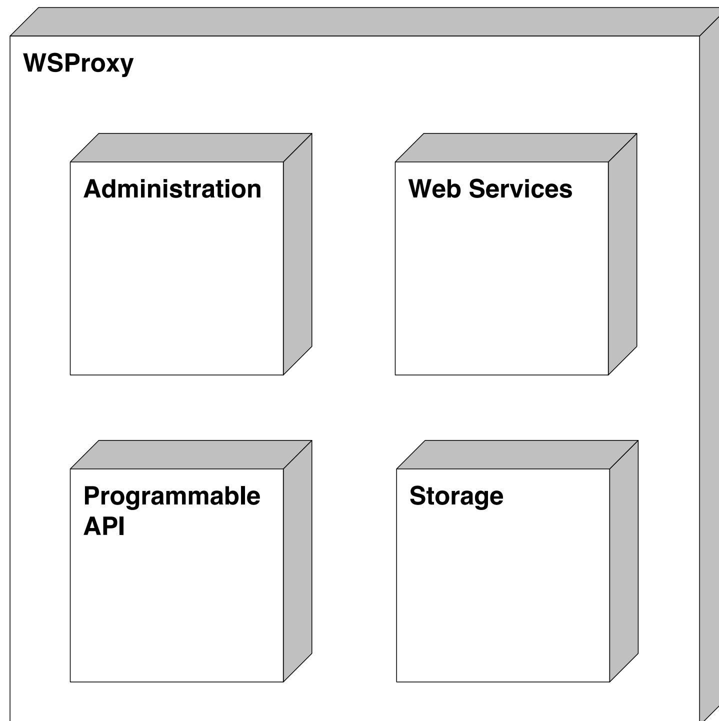


Figure 6. WSProxy architecture

There are four core components in a WSProxy application:

- **Administration** – This component is responsible for the configuration and administration of WSProxy. It has Web interface and can be accessed by a regular browser. Through the admin module, Web services can be registered with WSProxy, storage data can be manipulated or transit traffic examined. It gives access to a directory with information about other proxies.

- **Web Services** – This is the component that acts as a gateway to the WSDLs of all Web services registered with the proxy. This interface is used by the workflow engines to access the Web services functionality. It accepts SOAP messages on a standard HTTP port.

- **Storage** – This is the storage sub-system of WSProxy. It is used as a repository for referenced data, for storing system settings and captured transit traffic. It is implemented on top of a relational database.

- **Programmable API** – This is a SOAP Web service, hosted inside WSProxy, which gives programmable interface to part of the storage subsystem. It can be invoked by the Workflow engines when they need to manipulate referenced data. Primary use of the API is to delete unused references. However, it can also be used for data-flow optimizations by replicating referenced data among remote proxies.

5. Implementation

5.1. Administration component

In the administration component is concentrated the functionality for configuration and administration of WSProxy. The main responsibilities of the module are:

- Settings configuration and management
- Web Service Registration
- Connected proxies Registration

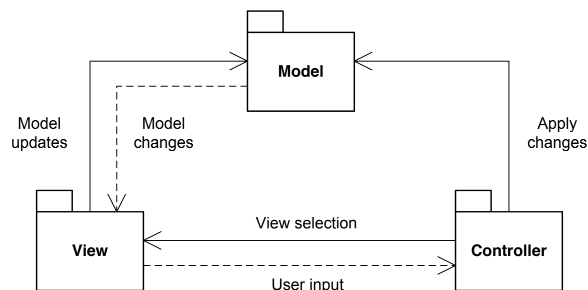


Figure 7. Model-View-Controller

With the aim of minimizing the dependencies in the module, the implementation follows the principles of Model-View-Controller design pattern [22]. The component functionality is divided into three logical components with strict separation of responsibilities.

- **View** – Takes care of visualization and provides forms for user input. It is implemented as a collection of Java Server Pages (JSP).

- **Model** – Contains the “business” logic. Provides persistence functionality for the user selections and input. The rules for the registration and modification of data are distributed among number of independent classes.

- **Controller** – Controls and validates the user input. Updates the model and selects the appropriate view (JSP). Implemented as a collection of Java Servlet classes.

The access to the administrator module is controlled by HTTP basic access authentication method. For each initial request from the client’s Web browser, the user should provide a user name and password. These user name and password are matched against a predefined administrator role: “proxy-admin”, stored in the application settings.

5.2. Web Services

This component implements the functionality for processing the messages coming from the Workflow Engine and the Web services. It exposes the WSDL interfaces of all registered with the proxy Web Services. Web Services component supports the styles of SOAP messages bindings – document/literal and rpc/literal. Services' WSDL definitions are parsed with the help of an external framework EasyWSDL [23]. EasyWSDL is a powerful tool that works with both WSDL 1.1 and WSDL 2.0. On the figure below are shown the main classes of Web Services component:

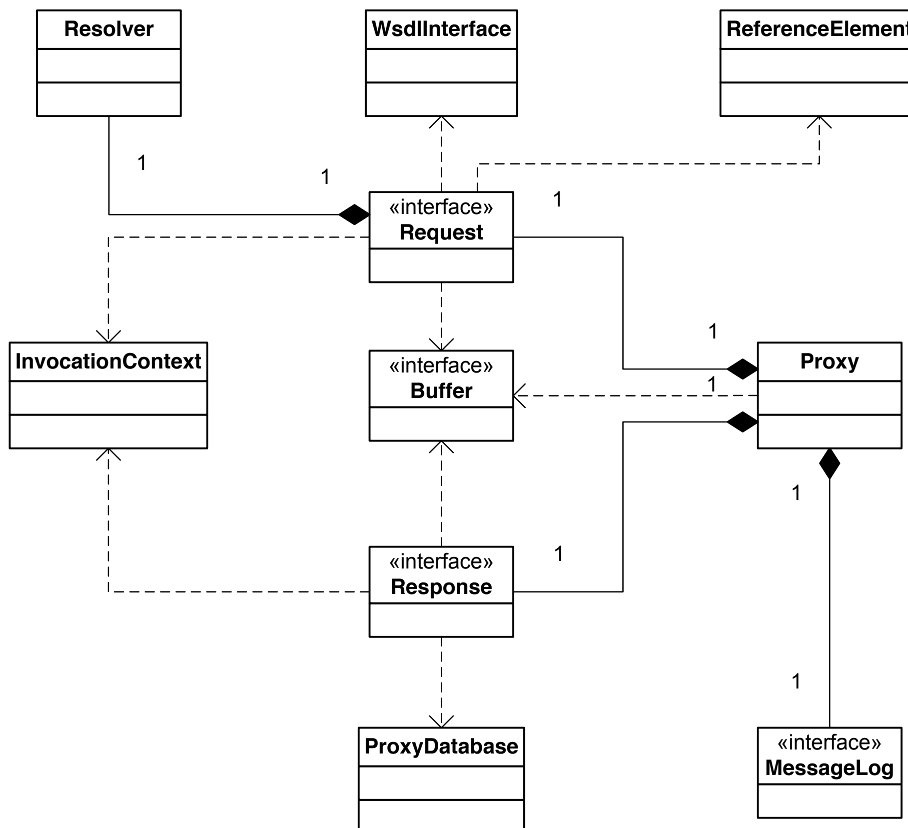


Figure 8. Main classes in Web Services

- **Resolver** – Classes that implement this interface are responsible for the reference resolution. It contains the logic for expanding the reference types to the actual data they point.
- **WsdlInterface** – Contains the parsed information about the interfaces (endpoints) of all registered Web Services' WSDL definitions.
- **ReferenceElement** – Abstracts the information about a reference type. It contains the XML element information extracted from the incoming requests. In this class is stored the actual data URL and identifier. The Resolver class uses this information to extract the data.
- **InvocationContext** – This class stores all information that is used between request and response. For example, the name of the request's operation is kept in the context.
- **Buffer** – This interface defines functionality for transit message persistence. It has two implementations that can be configured depending on the available resources. The first

implementation is the slower version that stores the messages on hard disk, while the second one is an in-memory buffer.

- **ProxyDatabase** – Facade of the application's data layer. It abstracts the access to the underlying data source. In the current implementation it is based on a local SQL database.
- **Request** – Abstraction of an SOAP request messages. It is implemented by two classes; the first is responsible for handling requests from messages with document style binding while the other is responsible for the rpc style messages.
- **Response** – Abstraction of an SOAP response message. It is implemented by two classes; one for the document bindings and the other for rpc style bindings.
- **Proxy** – Façade of the functionality for accepting and forwarding messages. The top class that organizes the work of the message processing and storing classes.
- **MessageLog** – Interface defining functionality for storing the transit traffic that passes via the proxy. If capture messages options is selected, class implementing this interfaces is responsible for the persistence of all incoming and outgoing messages.

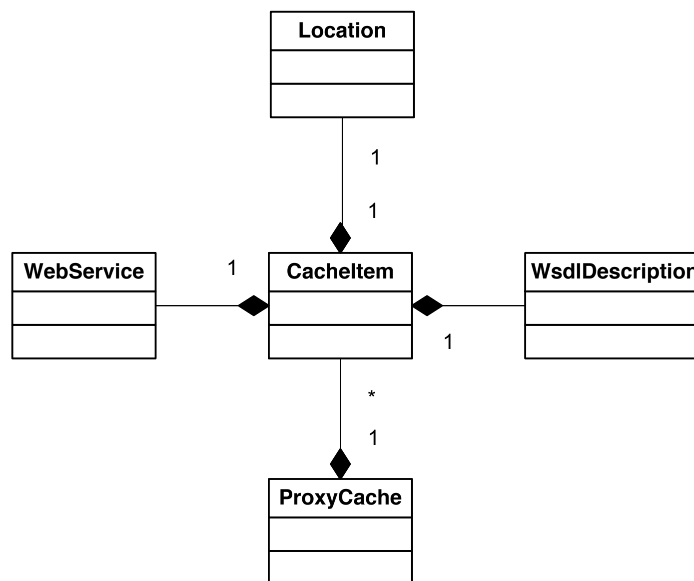


Figure 9. Proxy cache

The initial performance tests of the Web Service component showed that it is processing the messages way too slow. This was partially due to the fact that it routes large volumes of XML data. However, part of the performance penalties came from the constant database access needed to load the WSDL descriptions of the operations involved in the message exchange. Moreover, after the descriptions had been retrieved from the database, they had to be parsed by the EasyWSDL framework and only then loaded into memory. In order to solve this problem, a simple caching functionality was implemented. In this cache component, for each Web Service registered in WSProxy, is stored an item containing the WSDL description, the geographic location and the Web Services' metadata. The cache is implemented as a map data structure that associates each cache item with a Web Service identifier. It is a singleton class that is initialized by a "lazy" load style, when first accessed by clients.

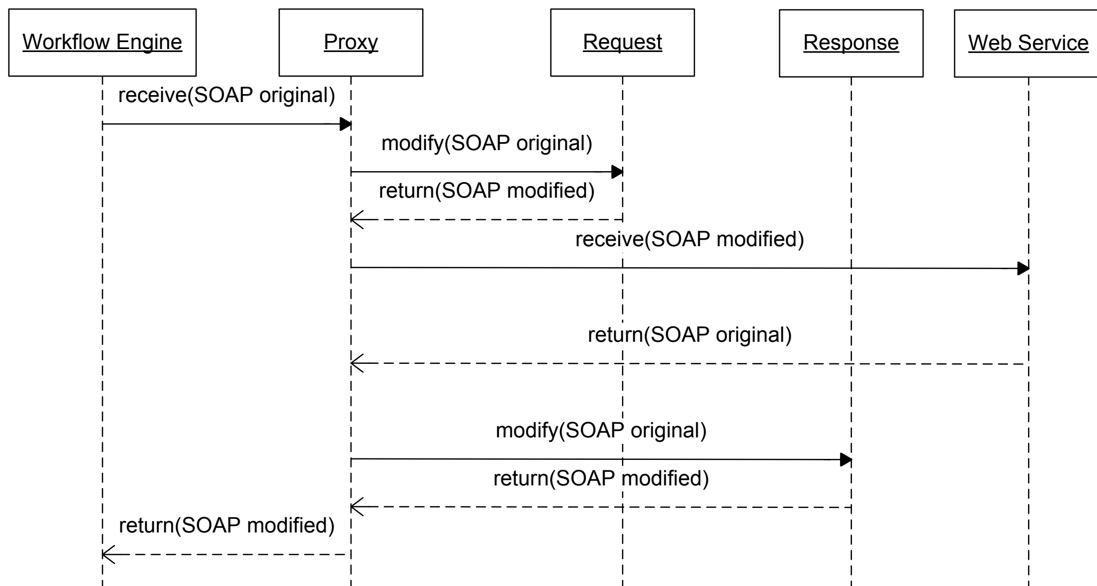


Figure 10. Message processing

On the figure above, is shown a simplified sequence diagram describing the algorithm for processing messages in the Web Services component.

When a message is received at the proxy, from the HTTP id request parameter is determined the Web Service identifier. By this identifier, the service description is retrieved from the cache and the type and binding style of the received message is established. For the request and response messages the style should be the same. Next, the proxy facade sends the received message the corresponding request object. The request object examines the request and, if the request contains referenced parameters, replaces all parameters with actual data. The modified request is returned to the proxy. The proxy sends the modified request to the actual Web Service and blocks until a response is received. The received service response is passed to the response object. The response object examines the response and, if the response is marked for referencing in the WSDL definition, replaces the payload data with a reference. The modified response is returned to the proxy. Finally, the proxy returns the modified response to the Workflow Engine.

All incoming messages that pass via the proxy are stored in temporary buffers. This is due to the fact that before making request to the corresponding Web Service, it is necessary for the message size to be known. Some HTTP servers provide functionality for automatic buffering of messages, but this is not a standard feature. Therefore, in order for the WSProxy to be as portable as possible, it was decided all incoming messages to be buffered on disk or in-memory.

The processed by WSProxy messages may be selected for capturing in the application settings. If so, all traffic passing via the proxy is recorded. Currently, the system offers functionality for logging the messages on the system console or to store them in the database. The stored messages can be examined from the administrator module.

```
  <?xml version="1.0" ?>
- <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  - <S:Body>
    - <ns2:Reference xmlns:ns2="http://www.ws.proxy.net/Schemas/Reference/v1"
        xmlns:ns3="http://common_types/test/services/v1"
        xmlns:ns4="http://consumer/">
      - <ns2:ref>
          <ns2:URL>http://127.0.0.1:8084</ns2:URL>
          <ns2:id>5</ns2:id>
          <ns2:operation>putData</ns2:operation>
          <ns2:ns>unused</ns2:ns>
          <ns2:loc>unused</ns2:loc>
        </ns2:ref>
      </ns2:Reference>
    </S:Body>
  </S:Envelope>
```

Listing 8. Message containing reference

On the figure above is shown an example of “SOAP original” (request in) message with reference parameter that points to data with identifier 5 stored on the local host. After this message was received and modified by the request object, the reference is resolved to the actual data.

```
- <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  - <S:Body>
    - <ns2:putData xmlns:ns2="http://common_types/test/services/v1"
        xmlns:ns3="http://provider/">
      - <ns2:data>
          <ns2:id>1</ns2:id>
          <ns2:num>0.1234</ns2:num>
        </ns2:data>
      - <ns2:data>
          <ns2:id>2</ns2:id>
          <ns2:num>1.1234</ns2:num>
        </ns2:data>
      - <ns2:data>

          ...

      - <ns2:data>
          <ns2:id>9999</ns2:id>
          <ns2:num>9998.123</ns2:num>
        </ns2:data>
      - <ns2:data>
          <ns2:id>10000</ns2:id>
          <ns2:num>9999.123</ns2:num>
        </ns2:data>
      </ns2:putData>
    </S:Body>
  </S:Envelope>
```

Listing 9. Message containing actual data

On the figure above is shown an example of “SOAP modified” (request out) message with resolved reference parameter. The data element <ns2:putData ... > contains an array of 10000 items.

5.3. Programmable API

This component is designed to allow access to the storage subsystem of WSProxy. The main idea behind it is to give Workflow engines the means to manipulate the referenced data, in particular to be able to delete the unused referenced data. It is developed as a SOAP Web Service, built on top the GlassFish Metro stack [24]. Besides the "housekeeping" functionality, the API can also be used for data-flow optimizations by replicating referenced data among remote proxies.

The API is hosted inside WSProxy and offers WSDL definition with three operations:

- Resolve – retrieves the referenced data into the Workflow Engine.
- Purge – deletes the referenced data.
- Replicate – retrieves the referenced data and makes a local copy. Creates a new reference.

```
 7  net.proxy.ws.schemas.reference.v1.ReferenceType ref = null;
 8  net.proxy.ws.schemas.reference.v1.ReferenceType refReplicated = null;
 9  try {
10      // generate test data
11      ProviderDocService_Service producer = new ProviderDocService_Service();
12      ref = producer.getProviderServicePort().getData(amount);
13
14      net.proxy.ws.schemas.reference.v1.Reference
15              refIn = new net.proxy.ws.schemas.reference.v1.Reference();
16      refIn.setRef(ref);
17      // use the test data
18      ConsumerDocService_Service consumer = new ConsumerDocService_Service();
19      consumer.getConsumerDocPort().putData(refIn);
20
21      // replicate the
22      refReplicated = refAPI.getProxyReferenceServicePort().replicate(ref);
23
24  } finally {
25
26      if (ref != null) {
27          // delete the test data
28          refAPI.getProxyReferenceServicePort().purge(ref);
29      }
30  } // finally
```

Listing 10. Example of Programmable API code

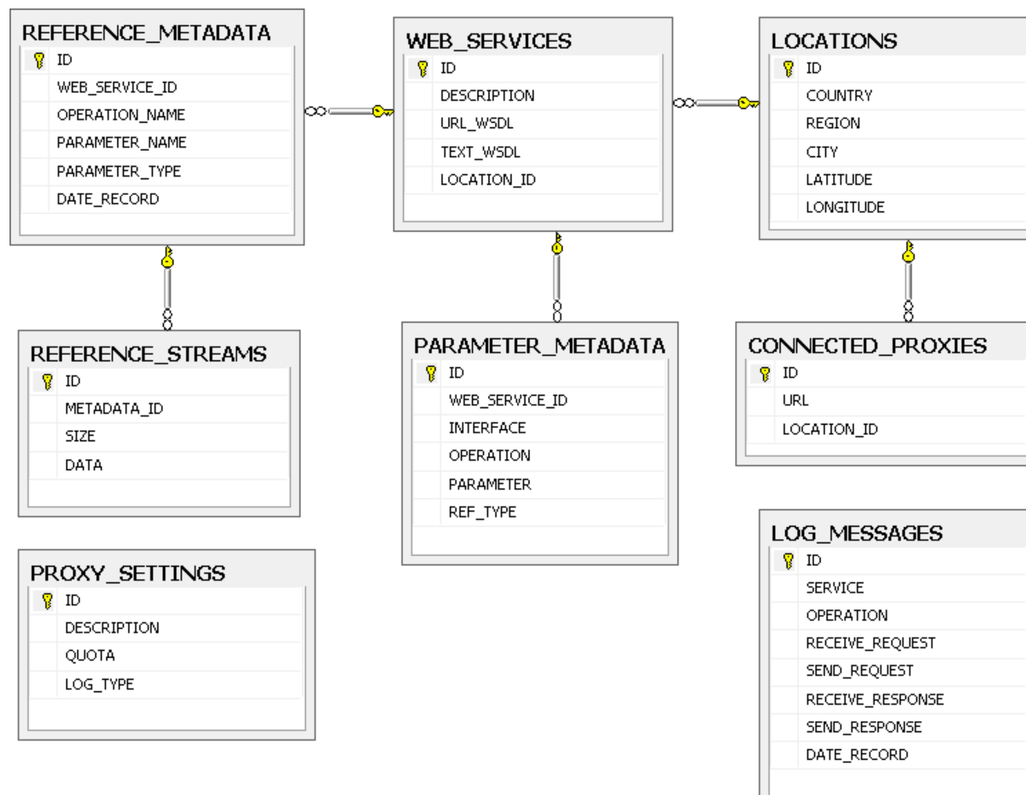
On the listing above is shown an example of a workflow that generates a test data that is returned as a reference. Then, this reference is passed to a consumer service. Next, the reference is replicated and, finally, at the end of the workflow on line 28, the reference is deleted.

Workflows can be implemented without the use of this API. However, in order to free the unused storage as quickly as possible, it is recommended that in the Workflow engine, at the end of the workflow, there is an explicit call for purging the referenced data.

5.4. Storage

Database Schema

Because of the relative simplicity of the database schema, it was decided that the WSProxy data storage component will be built on top of the Java Database Connectivity (JDBC) API. JDBC provides powerful, yet simple programming interface for accessing relational databases. The actual database is hosted on a local SQLite engine. SQLite is a self-contained, transactional SQL database that stores the whole information in a single disk file. It is a very popular and robust solution that provides local SQL database functionality.



Listing 11. WSProxy database schema

- **REFERENCE_METADATA** – Information about what generated a reference stream.
- **REFERENCE_STREAMS** – Related chunks of information. Stores referenced data.
- **WEB_SERVICES** – Information about Web Services – original URL, WSDL, description, etc.
- **LOCATIONS** – Stores location information about Web Services and remote proxies. **PARAMETER_METADATA** – Metadata about the original type of a referenced parameter.
- **CONNECTED_PROXIES** – Directory with URL of remote proxies.
- **LOG_MESSAGES** – Captured transit traffic.
- **PROXY_SETTINGS** – WSProxy application settings.

The information from WEB_SERVICES, LOCATIONS and PARAMETER_METADATA is stored in the WSProxy cache.

```
1  SELECT
2  m.ID,
3  ifnull(w.DESCRIPTION, "[Replicated]"),
4  m.OPERATION_NAME,
5  m.[PARAMETER_NAME],
6  m.DATE_RECORD,
7  s.STORAGE
8  FROM REFERENCE_METADATA AS m
9     INNER JOIN
10    (
11         SELECT METADATA_ID, sum(SIZE) AS STORAGE
12         FROM REFERENCE_STREAMS
13         GROUP BY METADATA_ID
14    ) AS s ON s.METADATA_ID = m.ID
15         LEFT OUTER JOIN WEB_SERVICES AS w ON m.WEB_SERVICE_ID = w.ID
16 ORDER BY m.ID DESC
```

Listing 12. Reference report query

On the listing above is shown the script used to generate reports about all referenced data stored on a proxy. The query is performed on several related table and returns aggregated information about the source and amount of storage taken by the streams.

Database classes

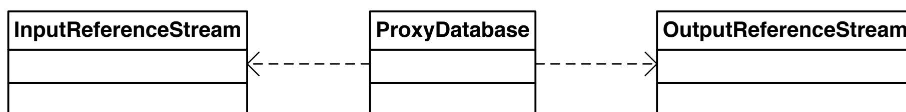


Figure 11. Main database classes

- **ProxyDatabase** - abstracts the calls to CRATE, SELECT, UPDATE, DELETE commands, and returns items that map to single rows in the SQL tables. On initialization, each instance of the class opens separate database sessions with the JDBC data source.
- **InputReferenceStream** – Handles the input reference data. Accepts the data byte by byte, or in byte arrays. The data is first stored in internal buffers and, after reaching a certain size, is flushed to the database. Maps to REFERENCE_METADATA and REFERENCE_STREAMS tables from the database schema.
- **OutputReferenceStream** – Database functionality for output referenced data. Reads the previously stored information.

The central class in the proxy's data access layer is ProxyDatabase. It is a Façade class that follows the principles of Table Data Gateway design pattern. [25]. ProxyDatabase offers a simple interface to the underlying SQL tables, moving data back and forth. Due to the fact that there are just several tables in the schema, only one gateway class that serves the whole database is implemented.

Because of the nature of processed data, the storing of referenced data is organized on a streaming principle. Instead of storing single records with large amount of data, the streaming database classes buffer the input and when certain threshold amount of data is accumulated, the buffers are flushed to the database. In this way very large amounts of data can be stored as a sequence of discrete streams with the same size. Keeping records with in this way avoiding the limitation on maximum row size.

5.5. Distance measurement

When registering Web services with the proxy, it is desirable for the administrator to have information about their approximate location. In order to accomplish that goal, several approaches could be taken. First, it is possible to use some kind of “ping” mechanisms to measure the round trip times and network delays between the proxy and the service. Another approach is to maintain statistical information about the delays with each service invocation. Over time when enough data is accumulated, this statistics can be used determine the optimal paths to route messages. However, in this thesis, it was decided to be used external services based on IP address geo location. With IP geo location, the IP address is used to deduce the actual host location by querying the regional Internet registries or databases provided by the local Internet service providers. In general, this is not the best approach; For example, it would not be accurate in cases of Mobile IPs, however, during the tests it proved to be very accurate. Some of the used services claim rate of accuracy over 90 percent. In all cases, even information about the continent on which is deployed the targeted web service is enough for the right proxy to be chosen.

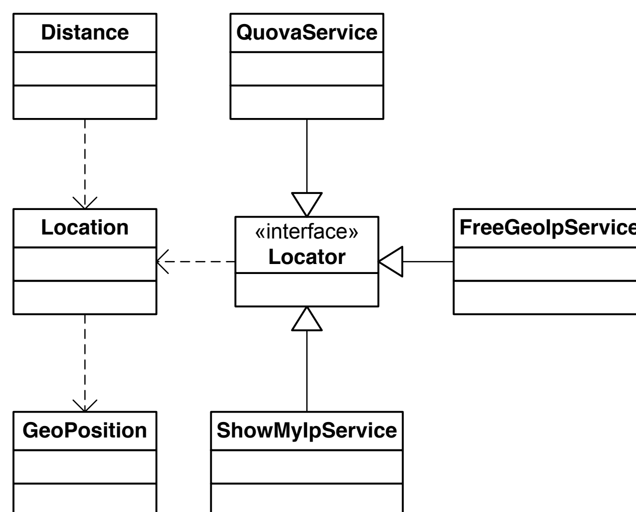


Figure 12. Distance measurement classes

- **Distance** – Class used to determine the estimated distance in kilometers between two hosts. If the target resides on the local host, the distance is always zero.
- **Location** – Contains the information the country, region, city and position.
- **GeoPosition** – Geographic position. Can calculate the distance between two geo positions objects, based on the Haversine formula[26].
- **Locator** – Interface defining functionality for finding the estimate location of a host.
- **QuovaService** – Implements Locator interface based on the API provided by Quova Neostar service [27]
- **FreeGeoIpService** – Implements Locator interface based on the API provided by Free IP geo location service [28]
- **ShowMyIpService** – Implements Locator interface based on the API provided by Show My IP service [29].

5.6. Quality assurance

During development, unit tests were designed for the core system functionality. Test class implementations for the message logging and resolving were also implemented. In addition to messages processing functionality, unit tests for the locator services were also developed. With these unit tests, referencing and dereferencing functionality document style binding can be tested without running the Web applications.

Besides unit tests, a console application hosting several workflows was developed. This application provides several end-to-end execution test cases for three workflow patterns – sequential, fan-in and fan-out. In addition, the same workflow patterns were implemented in Taverna.

6. Experiments

6.1. Introduction

6.1.1. Amazon EC2 instances

In order to confirm the performance benefits of the proposed solution, a set of experiments have been conducted. The test bed was organized across a network of Instances hosted on Amazon Elastic Compute Cloud (EC2) virtual machines. Amazon EC2 service allows users to host their applications on virtual machines deployed on six different geographic regions. For the purposes of this evaluation, instances from the micro family were selected. These micro instances are roughly equivalent to 613MB of memory with one virtual processing unit. Due to financial restrictions, the setup was limited to five micro instances running Apache Tomcat on 32-bit Linux operating system.

6.1.2. Regions

The regions of the instances were selected with the aim to be at such distances that they will allow for a wide area network (WAN) to be formed.

- **Ireland** - One instance was deployed in the west of the European Union in Ireland.
- **US East 1, 2, 3** – Three instances was deployed on the East Coast of the United States. These instances are geographically very close to each other and may be considered local.
- **US West** - The last instance was deployed on the West Coast of the United States.
- **Workflow** - The workflow engine was situated in Edinburgh, Scotland, in relative proximity to the instance in Ireland.

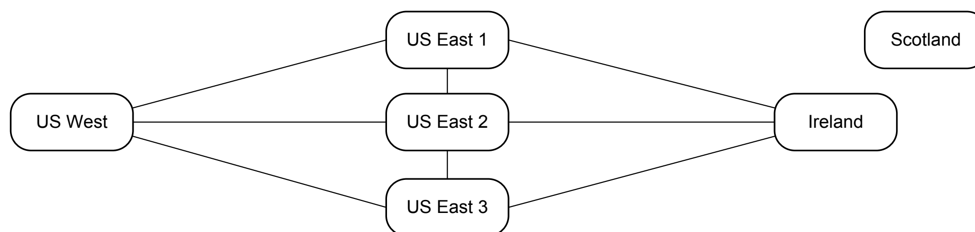


Figure 13. Geographical positioning of the instances

The instance deployment configuration is similar to the geometrical shape of the figure shown above.

6.1.3. Workflow engine

Taverna Workflow Management System was chosen to host and execute the test workflows. Taverna [7] is an open-source and domain independent Workflow engine specifically design to execute scientific workflows. Taverna has superb functionality for designing and executing workflows, rich GUI and features for capturing and examining intermediate results. This functionality, however, comes at a price. Workflows executed in Taverna incur performance penalty because of the additional non-workflow related computations. Therefore, in order to provide an accurate base for evaluation and comparison, all non-proxy workflows were also implemented in a Java console application. These workflows are logically equivalent to the ones implemented in Taverna.

The workflow engines were run on a single Windows machine with processing power and storage similar to those of the Amazon micro instances.

6.1.4. Test Web Services

Although the WSProxy solution can be used with regular Web services deployed on the Internet, in order to ensure stable performance and availability during tests, it was decided that two new test Web services will be implemented and deployed:

- **Producer** - Web Service designed to produce test data of given size. It exposes only one operation – getData that takes as a parameter size and returns an array of two element structures. Parameter of size 15000 is roughly equivalent to 1MB of XML data transfer.
- **Consumer** - Web Service designed to consume test data. It exposes only one operation - putData that takes as a parameter the array generated by Producer Web Service. The operation returns integer value with the size of the "consumed" array.

6.1.5. Taverna workflow elements

The blue elements on all Taverna workflow figures are input and output respectively. The s_1 to s_4 are XML splitter elements used to prepare the inputs and outputs in cases of parameter mismatch. These elements are not related to the business logic of the workflow and take insignificant time to execute, therefore they may be disregarded. The green elements are the central pieces of the workflow. They represent the producer-consumer Web Services participating in the workflow.

6.1.6. Measurements

In order to take into account the affect of the variable amount of data, all experiments were measured with three different messages sizes:

- **Small** – 15 element array of data structures, equivalent to a data transfer of 1 Kilobyte.
- **Medium** – 1500 element array of data structures, equivalent to a transfer of 100 Kilobytes.
- **Large** – 15000 element array of data structures, equivalent to a data transfer of 1 Megabyte.

For each scenario and message size, the experiments were executed in two versions - with proxy and without proxy. Each test run was repeated 10 times and the average values from the two versions were calculated.

6.2. Scenarios

The scenario patterns are based on those described in [5]. These workflow patterns are influenced by Montage, workflow common to many scientific applications. Three distinct types of scenarios were evaluated:

- **Sequential** – The test web services are ordered in sequence, where the data produced from one service is passed to the next service in the sequence: Producer –> Consumer.
- **Fan-in** – Simulates workflows in which the data from multiple sources is funneled to a single destination: Many Producers –> One Consumer.
- **Fan-out** – Simulates workflows in which the data from a single source is sent to multiple destinations: One Producer –> Many Consumers.

In addition, it was taken into account that the test results will be affected to large extend by the location of the proxy. In order to evaluate the effect of the proxy deployment, the same pattern scenarios (Sequential, Fan-in, and Fan-out) were repeated with the proxy deployed on a different instance, once in the United States and once in Ireland.

6.2.1. Sequential scenario

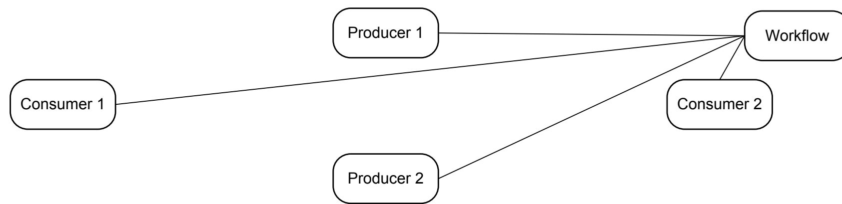


Figure 14. Sequential orchestration

On the figure above is shown fully centralized sequential scenario. All test data passes through the Workflow node located in Scotland.

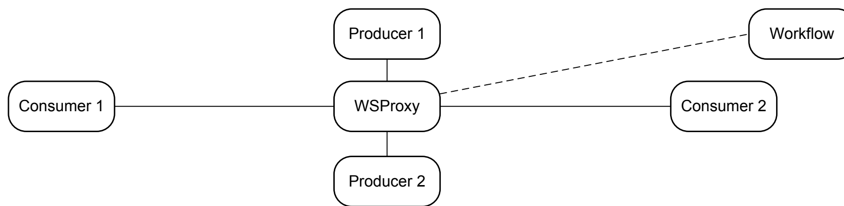


Figure 15. Sequential orchestration with Proxy in US

On the figure above is shown optimized sequential scenario with a single proxy deployed in the middle. With dashed line is shown the communication link between the workflow engine and the proxy. On that link are flowing only control messages and references. Consequently, the heavy traffic stays between the proxy and the Web services.

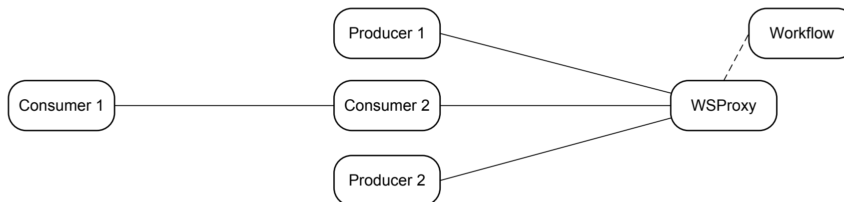


Figure 16. Sequential with proxy in Ireland

On the figure above is shown optimized sequential scenario with the proxy deployed in Ireland. This pattern has less optimal dataflow than the one with the proxy deployed in US, however the workflow remains same, only locations change.

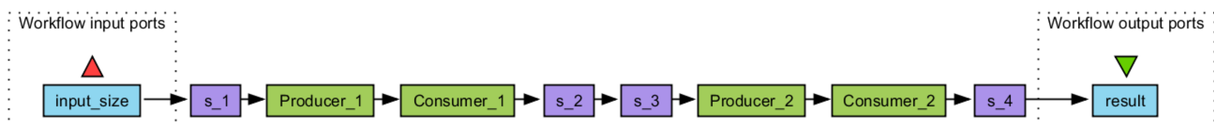


Figure 17. Sequential workflow in Taverna [7]

In the workflow above, Producer 1 Web service generates test data which is passed to Consumer 1. Next, Producer 2 generates test data which in turn is passed to Consumer 2.

The results for the sequential scenarios are shown below. Lower values mean better performance.

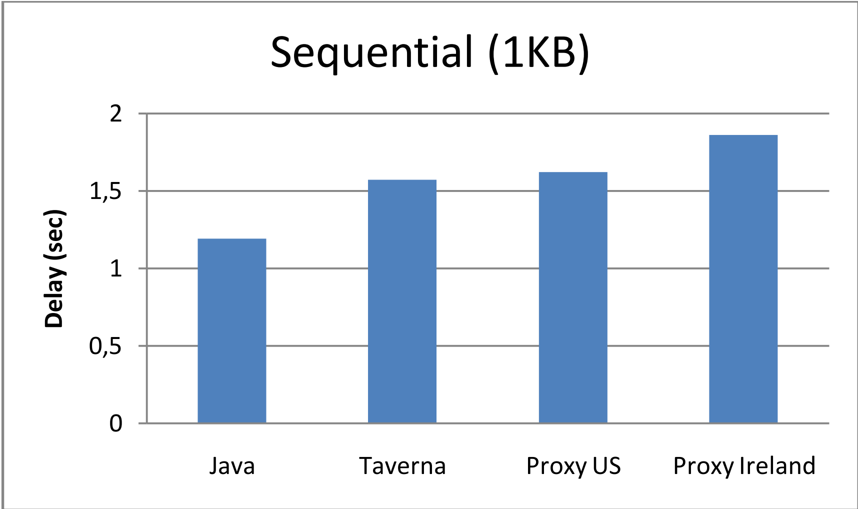


Figure 18. Results sequential scenario with 1KB

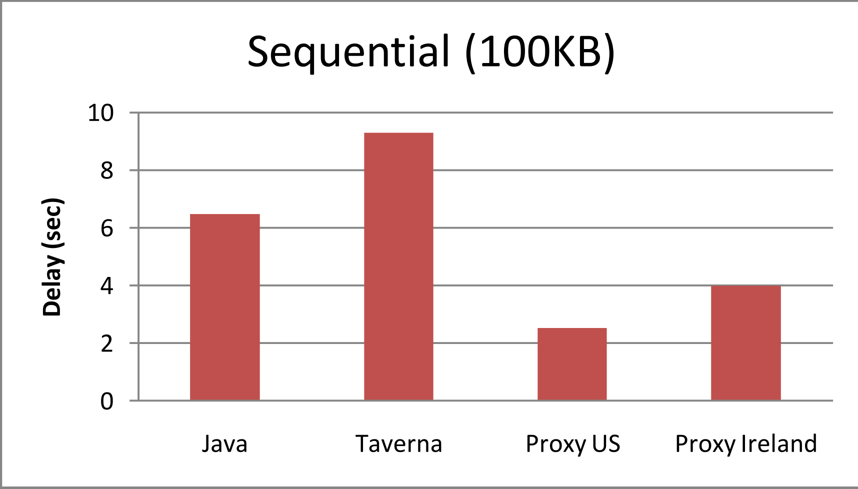


Figure 19. Results sequential scenario with 100KB

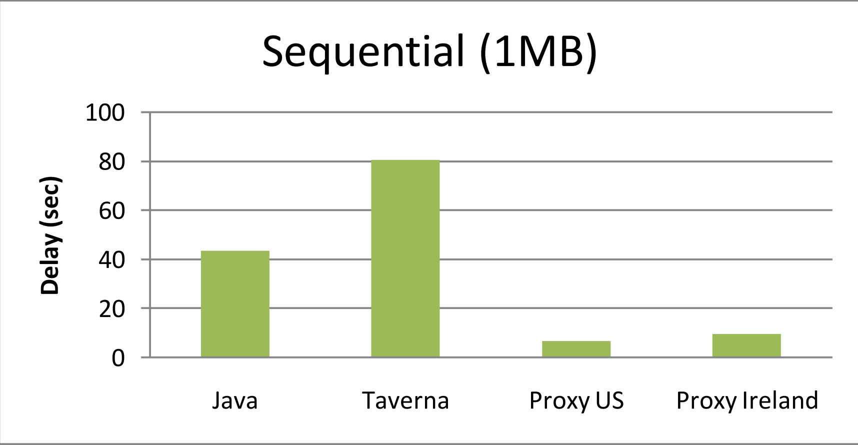


Figure 20. Results sequential scenario with 1MB

6.2.2. Fan-in scenario

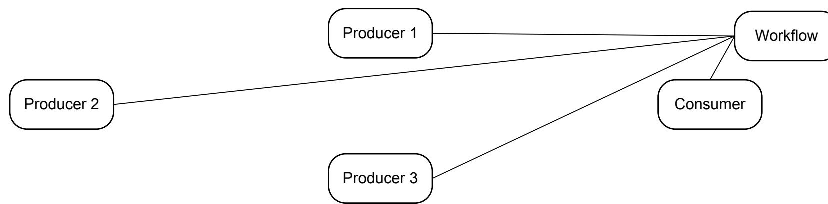


Figure 21. Fan-in orchestration

On the figure above is shown fully centralized fan-in scenario. All test data passes through the Workflow node located in Scotland.

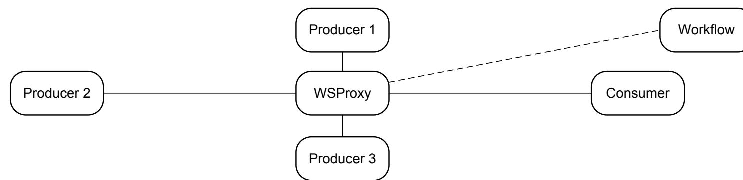


Figure 22. Fan-in orchestration with Proxy in US

On the figure above is shown optimized fan-in scenario with a single proxy deployed in the middle. With dashed line is shown the communication link between the workflow engine and the proxy. On that link are flowing only control messages and references. Consequently, the heavy traffic stays between the proxy and the Web services.

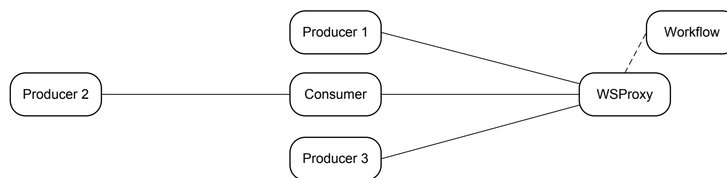


Figure 23. Fan-in orchestration with Proxy in Ireland

On the figure above is shown optimized sequential scenario with the proxy deployed in Ireland. This pattern has less optimal dataflow than the one with the proxy deployed in US, however the workflow remains same, only locations change.

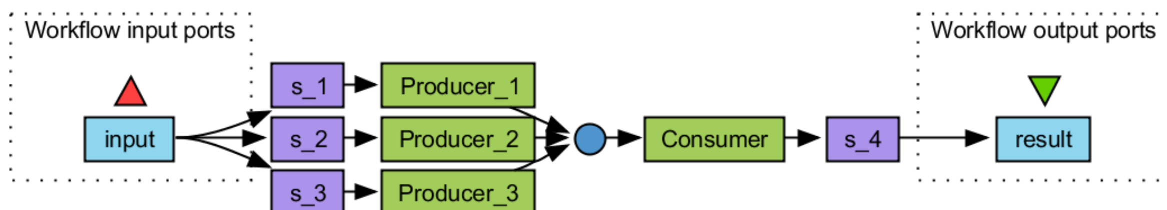


Figure 24. Fan-in workflow in Taverna [7]

In the workflow above, Producer 1 to 3 generate test data in parallel. These data are merged into a list and then passed to the consumer Web Service.

The results for the fan-in scenarios are shown below. Lower values mean better performance.

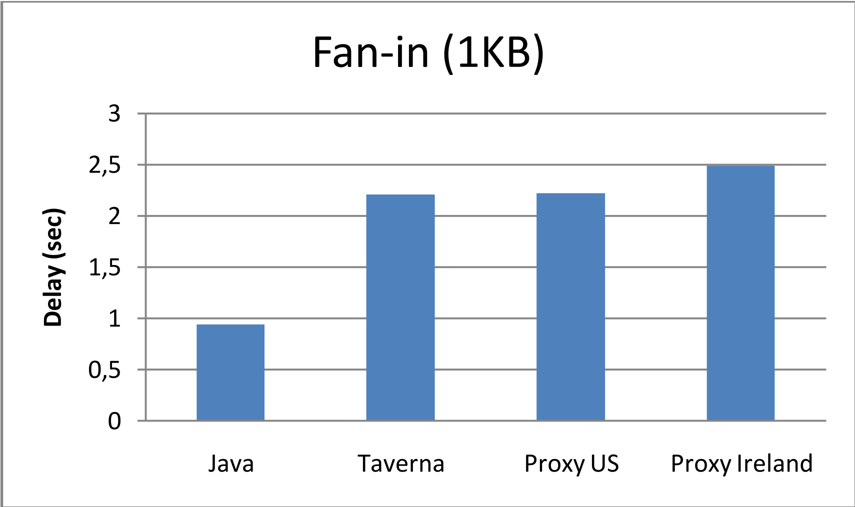


Figure 25. Results fan-in scenario with 1KB

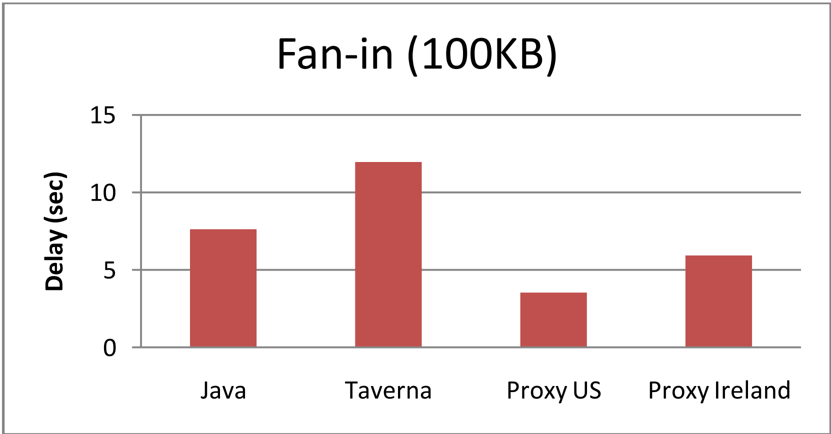


Figure 26. Results fan-in scenario with 100KB

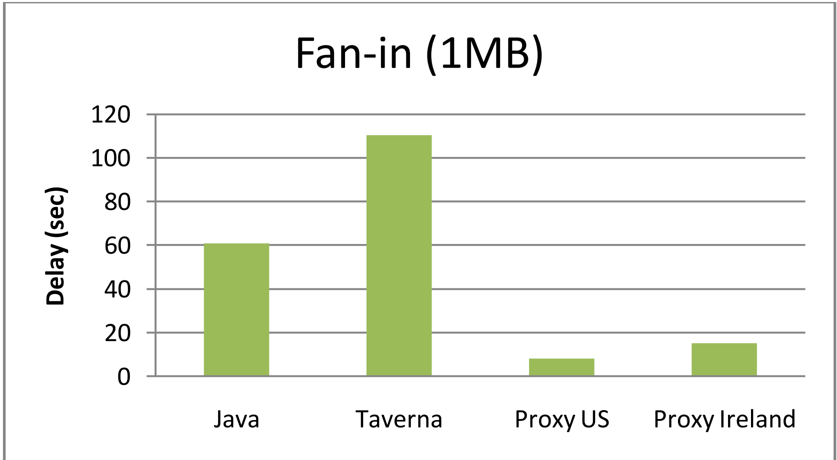


Figure 27. Results fan-in scenario with 1MB

6.2.3. Fan-out scenario

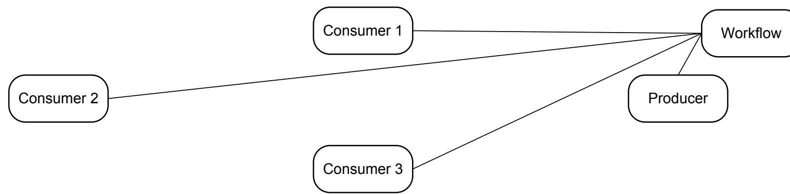


Figure 28. Fan-out orchestration

On the figure above is shown fully centralized fan-out scenario. All test data passes through the Workflow node located in Scotland.

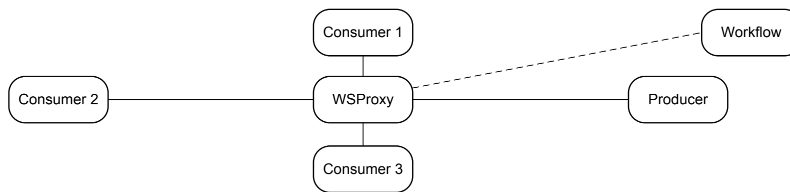


Figure 29. Fan-out orchestration with proxy in US

On the figure above is shown optimized sequential scenario with a single proxy deployed in the middle. With dashed line is shown the communication link between the workflow engine and the proxy. On that link are flowing only control messages and references. Consequently, the heavy traffic stays between the proxy and the Web services.

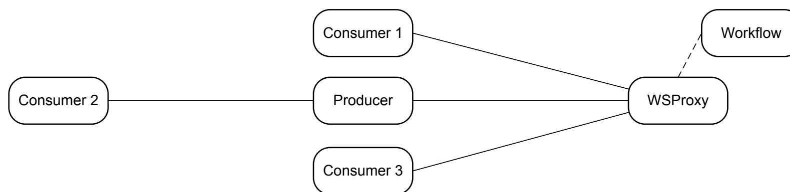


Figure 30. Fan-out orchestration with proxy in Ireland

On the figure above is shown optimized fan-out scenario with the proxy deployed in Ireland. This pattern has less optimal dataflow than the one with the proxy deployed in US, however the workflow remains same, only locations change.

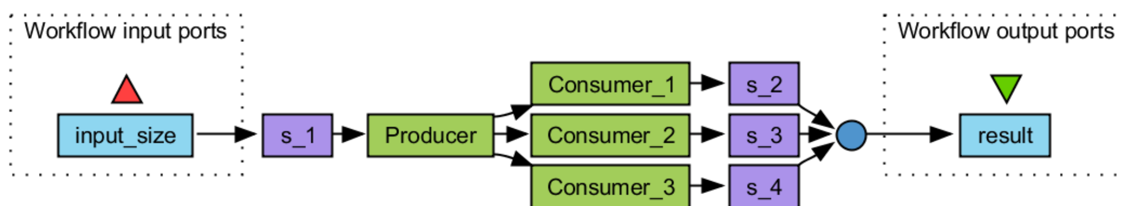


Figure 31. Fan-out workflow in Taverna [7]

In the workflow above, Producer Web Service generates test data which is send in parallel to Consumer 1 to 3 Web Service. Next, the consumer's results are joined at the finish.

The results for the fan-out scenarios are shown below. Lower values mean better performance.

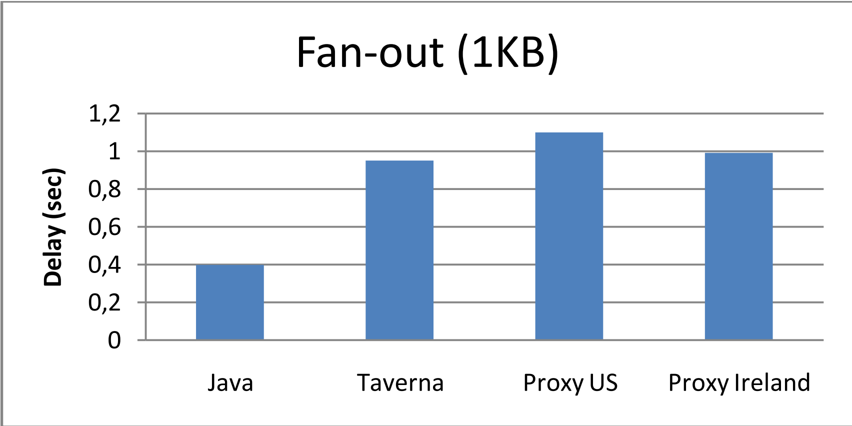


Figure 32. Results fan-out scenario with 1KB

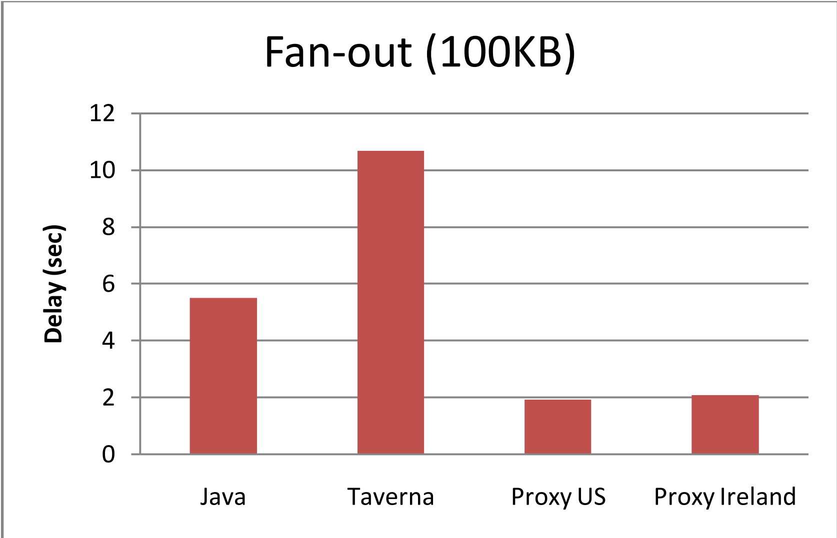


Figure 33. Results fan-out scenario with 100KB

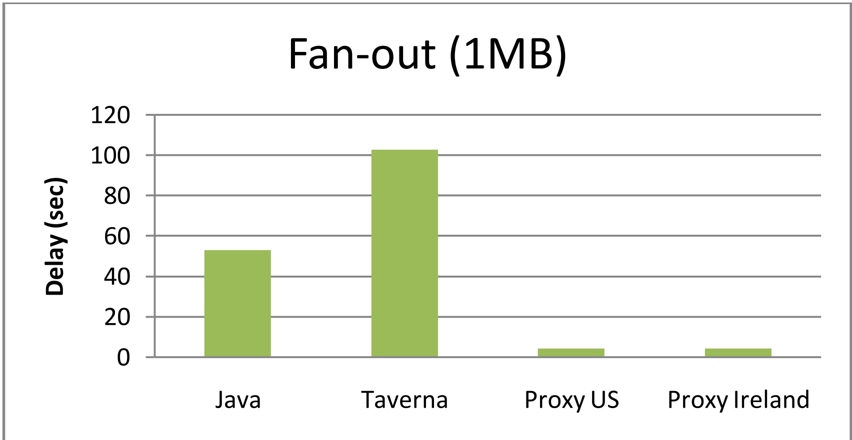


Figure 34. Results fan-out scenario with 1MB

6.3. Performance evaluation

In order to validate the hypothesis that dataflow optimized by proxies will be beneficial to orchestrated workflows, the experimental data from the previous sections was aggregated and evaluated. For each scenario and data size, the data is represented as improvement ratios. The ratios are measured as the total delay in one pattern divided over the total delay in another. Values greater than one indicate improvement.

6.3.1. Evaluation of workflows implemented in Java and Taverna

Sequence	1KB	100KB	1MB
Java over Taverna	1,32	1,44	1,85
Fan-in	**1KB**	**100KB**	**1MB**
Java over Taverna	2,03	1,57	1,81
Fan-out	**1KB**	**100KB**	**1MB**
Java over Taverna	2,37	1,94	1,94

Table 1. Improvement ratios for Java over Taverna workflows. Value greater than one is better

On the figures above are shown the improvement ratios between Java orchestration and Taverna orchestration. For all scenarios, the same workflows implemented in Java are performing better than those implemented in Taverna. The improvement ratios range from 1.32 for the smaller messages and increase up to 1.94 for the largest messages. The better performance of Java workflows is due to the fact that Taverna's carries out additional computations related to animated GUI, storing intermediate results, etc. These computations are not directly related to the workflow logic but consume resources, consequently, the same Taverna workflows implemented in a Java console applications complete faster.

In order to negate the detrimental effect of non-workflow related computations, it was decided that Java workflows will be selected as a model for non-proxy orchestration. This more conservative approach will give clearer picture of the improvements in the scenarios in which proxies store and forward data.

6.3.2. Evaluation of proxy scenarios

Sequence	1KB	100KB	1MB
Proxy Ireland over Java	0,64	1,64	4,56
Proxy US over Java	0,73	2,56	6,32
Fan-in	**1KB**	**100KB**	**1MB**
Proxy Ireland over Java	0,39	1,31	4,06
Proxy US over Java	0,43	2,20	7,81
Fan-out	**1KB**	**100KB**	**1MB**
Proxy Ireland over Java	0,41	2,65	12,53
Proxy US over Java	0,36	2,85	13,40

Table 2. Improvement ratios for the proxy scenarios. Value greater than one is better

For small messages (1KB) all tests show that introducing proxy will not be beneficial. On the contrary, it is evident that there is performance degradation when a proxy is used. This result is explained by the specifics of the exchanged messages. In each message, no matter its size, there is an overhead incurred by SOAP XML and HTTP/TCP stacks. The experiments show that for small payloads of size 1KB the overhead is significant. In the sequential scenario, for example, the total number of messages exchanged in non-proxy scenario is four compared to eight with proxy deployed in US. From the eight messages, four are messages carrying references, and the rest four messages are containing the actual data. However, all messages are of roughly equal size, which means that there is twice as more network traffic and message exchange.

Conversely, with the increase of the message size, the benefits of using proxies become more evident. When the message size is increased to over 100KB, across all scenarios is observed improvement from 1.64 to 13.40 times. This is due to two major reasons. First, the message overhead becomes less significant with the increase of the payload. Second, more importantly, the orchestration nodes perform very costly parsing, binding (XML to Java object) and validation of the SOAP traffic. Consequently, the processing time of messages increases proportionally to the message size. On the other hand, because the proxy does not perform XML data binding or validation with the message size increase the performance of the proxy scenarios also increases.

Proxy deployment also plays significant role in the performance improvement. Across all scenarios, when the proxy is closer to the participating Web services, the performance gains are higher. In sequential and fan-in scenarios, when the proxy is deployed in the US, the workflow is executed about 40 - 50% faster than the workflow with proxy deployed in Ireland. This is due to the fact that the large data transfers stay in the US and only control messages flow across the Atlantic.

With respect to the different scenarios, the best performance gains are achieved in fan-out workflows. The major difference between fan-out and the other scenarios is that fan-out workflows are characterized by data upload. In fan-out, the data downloaded from one source is simultaneously uploaded to many destinations. The workflow ends when the last upload is finished. Evidently, the proxy configurations handle upload much better than the centrally orchestrated ones.

7. Future Work

There several directions in which the work in this thesis could be extended.

7.1. Improvements in functionality

- Security – For the moment, WSProxy implements only basic elements of security. Access to the admin functionality is controlled by user name and password. However, the access to the Programmable API is unrestricted. All data could be read or modified without restriction or audit trail.
- Functionality for encoded style SOAP – At the present, WSProxy can process SOAP messages in document/literal and rpc/literal styles. However, the EasyWSDL framework has a limitation that prevents it from modifying rpc/encoded definitions. Probably, for the future, it would for the best if EasyWSDL is replaced with more robust framework.
- Transformations on the proxy – Currently, the data extracted from the SOAP messages is stored on the WSProxy as it is. This, however, means that the signatures of the consumer and producer services have to be the same – same name and data type. In some cases, it will be very useful to have some means of modifying the stored data - XSLT transformation.
- Improvements in the XML parsing – The existing XML parsing functionality is a custom developed parser similar to SAX. However, for richer functionality which will allow better XML handling, it is recommended that some standard JAVA parser is used, perhaps some StAX implementation.
- Load balancing of proxies – A good extension would be functionality for load balancing of the invocations of referenced Web services.

7.2. Research and Experiments

- Experiments with multiple proxies – All experiments in this thesis involve one proxy configurations. It will be very interesting, however, to build some scenarios with multiple proxies. For example, one proxy for Web services deployed in the United States and one proxy for services deployed in Europe. This will give opportunity to test “replicate” operation of the Programmable API, which moves data across proxies.
- Experiments with real-life Web services – Workflows with existing Web services should be designed and executed, in order for “real-life” scenarios to be evaluated.
- Experiments with “transformer” Web Services – Because the proxy does not offer XML transforming functionality, for the moment as a work around “transformer” Web Services could be used to modify the input/output data. These services can be deployed locally to the proxy and when a client wishes to change the input or output formats, she could invoke the “transformer” services.
- Research into distance measurement – As the experiments have shown, the deployment location of a proxy plays crucial role. At the moment, IP geo location is used to measure the distance between proxies and Web services. However, other techniques may be investigated such as network distance measurement tools and statistics on round-trip-times of previous message traffic.

8. Conclusions

A key problem with orchestrated Web services – the central node bottleneck, was identified and solution has been proposed. A working prototype of the solution – Workflow Speedup Proxy (WSProxy), has been designed and implemented. The architecture and development of the software were examined. Series of experiments based on workflows implemented both in Java and Taverna were conducted. The results of these experiments show that, for some cases, introducing proxy to the workflow is not practical. In particular, with small message sizes of about 1KB there is significant performance degradation of more than 50%. However, with larger messages the benefits of introducing WSProxy became evident - workflow improvement of over 10 times in the fan-out scenario was observed. Possible future work and improvements, such as more elaborate experiments and better WSDL operation support, were pointed out at the end of the work.

Appendix A: Installation

Projects

The solution is developed in Java, NetBeans 6.9.1 IDE and uses Apache Tomcat 6.0.26.

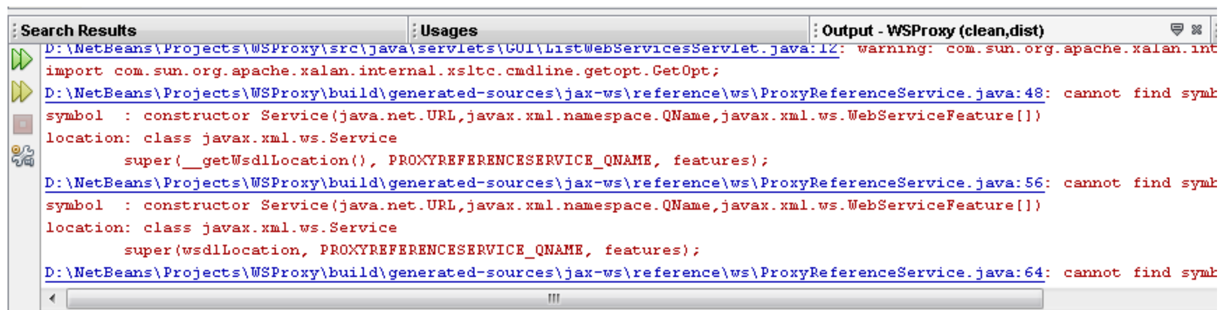
- **TestServices** – Web Application that contains test consumer/producer services. Each service has two versions, one for document style messages and one for rpc style.
- **WorkflowEngine** – console application with several test workflows.
- **WSProxy** –Web application, JSP, Servlets and a Web service.

External dependencies of WSProxy:

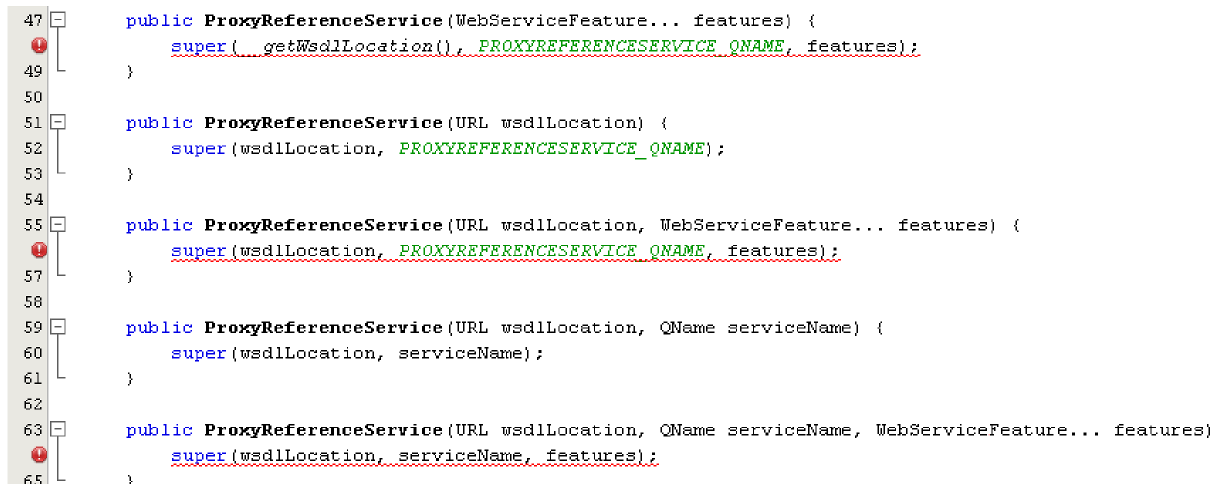
- EasyWSDL 2.1: http://easywsdl.ow2.org/download.html
- SQLite 3.7.2: http://mvnrepository.com/artifact/org.xerial/sqlite-jdbc/3.7.2

Build instructions

During the build of WSProxy and TestServices, the IDE will produce several errors in the automatically generated code for Web services:



These errors are result of inconsistencies in the NetBeans implementation of the JAX-WS framework. They are caused by constructors with parameters containing WebServiceFeature types. All such constructors should be deleted.



On the screenshot above, the three constructors declared on lines 47, 55 and 63 should be removed.

Security settings

Access to the admin pages of WSProxy is restricted by basic HTTP authentication mechanism. WSProxy allows only authenticated uses with assigned role "proxy-manager". In Apache Tomcat, the user roles and passwords are contained in tomcat-users.xml file located in the **conf** folder.

```
  limitations under the License.
-->
<tomcat-users>
  <role rolename="proxy-manager"/>
  <user username="admin" password="pass" roles="admin, manager, proxy-manager"/>
</tomcat-users>
```

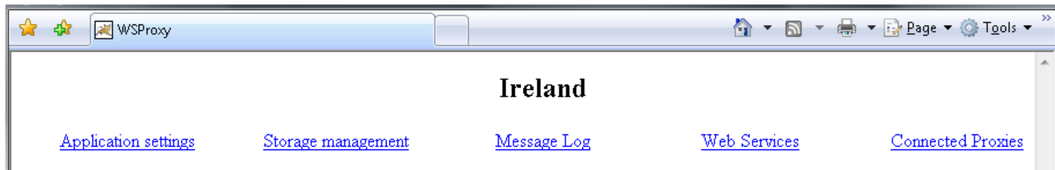
Note: Be sure to restart the server after changing the user credentials.

Deployment

The application is contained in a single WAR file that should be placed in the Tomcat's **webapps** folder. The database of the application is automatically created in Tomcat's **bin** folder.

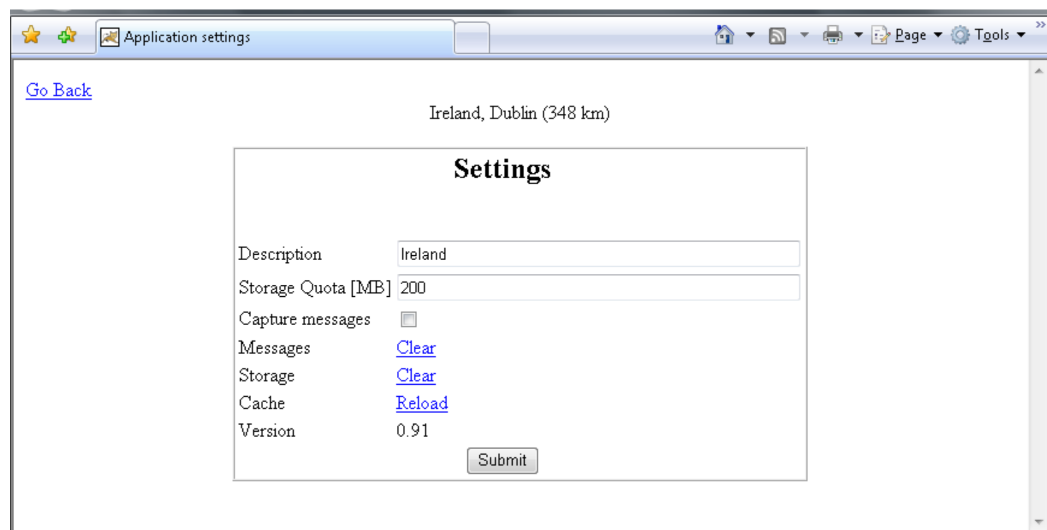
Appendix B: WSProxy Web interface

Main page



- **Application settings** – Changing the applications settings.
- **Storage management** – View/Delete stored reference data.
- **Message log** – View the captured message traffic.
- **Web Services** – Registering and configuration of Web Services.
- **Connected Proxies** – Directory with other WSProxy applications

Application settings



- **Description** – Short textual description of the application. This description shows on the main page of the application. It is also shown as a description in the list with connected proxies.
- **Storage Quota** – Upper limit in Megabytes of the allowed storage on the proxy. If any application tries to store more data than the limit, the proxy returns OutOfStorageException.
- **Capture messages** – Turns On/Off the traffic messages capture. If On, all messages that pass through the proxy will be recorded and can be examined. The option is turned off by default.
- **Messages Clear** – Clicking on the hyperlink will delete all captured transit traffic.
- **Storage Clear** – Deletes all stored reference data. This options should be used with caution because may result in invalid references used by the WSProxy clients.
- **Cache Reload** – Reloads all information from the database. Should be used if the database changes while the application is running. For example, if WSDL definition is updated.
- **Version** – Current version of the application. Increased with every software change.

Storage management

Go Back

Reference data

Used storage 4 %

Service description	Operation	Parameter	Date and time	Storage (KB)	Action
Producer 3	getData	Response	2011-08-27 15:15:25	1026.4	View / Delete
Producer 2	getData	Response	2011-08-27 15:15:24	1026.4	View / Delete
Producer 1	getData	Response	2011-08-27 15:15:24	1026.4	View / Delete

Report with all reference data stored on the proxy, sorted by date and time in descending order.

- **Service description** – Textual description of the service that generated the data.
- **Operation** - Name of the operation that generated the data.
- **Parameter** – Name of the parameter that generated the data.
- **Data and time** – Date and time of the date generation.
- **Storage** – Shows how much space is taken by the data, in Kilobytes.
- **Action** – View or delete the reference data item.

Message log

Go Back

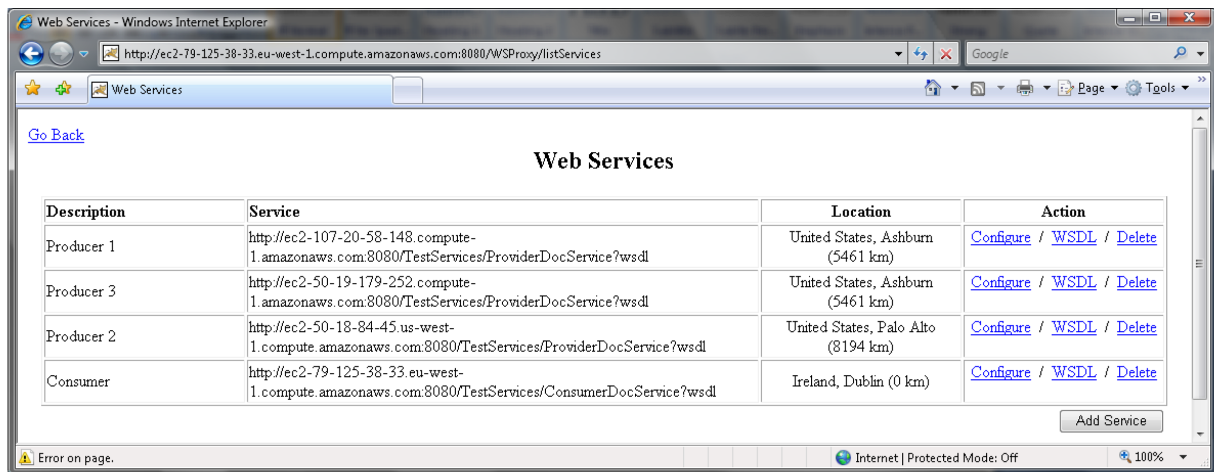
Message Log

Service	Operation	request in	request out	response in	response out	Date and time
Consumer	putData	Message	Message	Message	Message	2011-08-28 17:46:07
Consumer	putData	Message	Message	Message	Message	2011-08-28 17:46:06

Report with all captured messages, sorted by date and time in descending order. Row from the report contains the four messages involved in one operation invocation.

- **Service** – Textual description of the message's service.
- **Operation** – Name of message's operation.
- **Request in** – The incoming workflow request.
- **Request out** – Modified by the proxy request sent to the Web service.
- **Response in** – Response received from the Web service.
- **Response out** - Modified by the proxy response sent to the workflow engine.
- **Date and time** - Date and time of message capture.

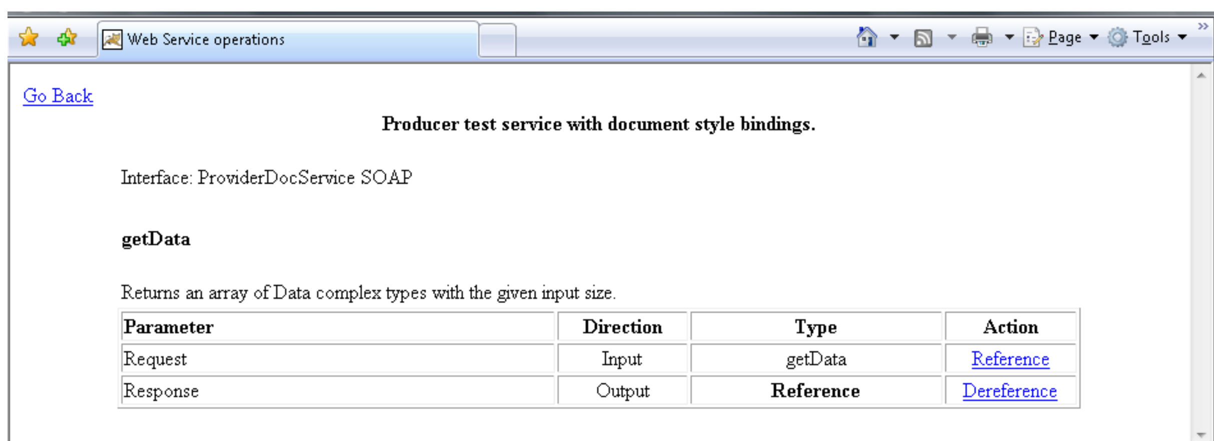
Web Services



Form for registration and configuration of Web services.

- **Description** – Description of a Web service, shown in the reports.
- **Service** – URL of the original Web service WSDL.
- **Location** – Estimated geographical location of the Web service.
- **Action, Configure** – Referencing / Dereferencing operation parameters.
- **Action, WSDL** – Displays the content of the modified WSDL definition. This is the WSDL that should be used in workflows.
- **Action, Delete** – Removes all information about a Web service – definitions, messages, etc.

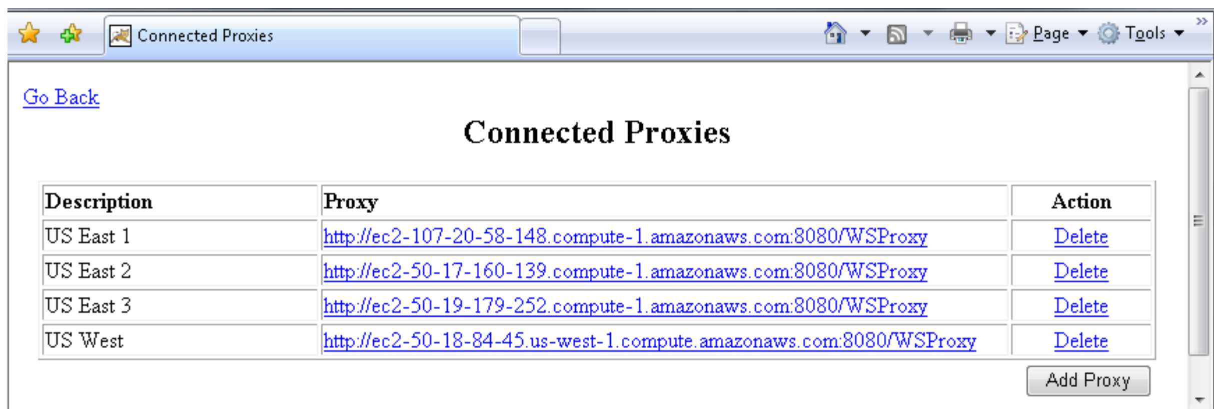
Action, Configure



Form for selecting Web service's operation parameter for referencing.

When a parameter is selected of referencing, the type names become bolded **Reference.** If WSProxy encounters message with such return type, it stores the contained data and replaces the parameter with a reference. If WSProxy encounters message with input type reference, it replaces the reference with the actual data.

Connected Proxies



Used for quick navigation across a network of proxies.

- **Description** – Textual description of the proxy. Retrieved from the connected proxy's application settings.
- **Proxy** – URL of the connected proxy.
- **Action, Delete** – Removes the proxy from the list.

References

1. MSDN. *XML Web Services Basics*. 25.06.2011]; Available from: http://msdn.microsoft.com/en-us/library/ms996507.aspx.
2. Sommerville, I., *Software engineering*. 9th ed. 2011, Boston: Pearson. xv, 773 p.
3. WorkflowManagementCoalition. *The Workflow Reference Model*. 10.07.2011]; Available from: http://www.wfmc.org/standards/docs/tc003v11.pdf.
4. Binder, W., I. Constantinescu, and B. Faltings, *Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration.* Int. J. High Perform. Comput. Netw., 2009. **6**(1): p. 81-90.
5. Barker, A., J.B. Weissman, and J.I. Hemert, *The Circulate architecture: avoiding workflow bottlenecks caused by centralised orchestration.* Cluster Computing, 2009. **12**(2): p. 221-235.
6. UniversityOfManchester. *Taverna Data Proxy*. 10.07.2011]; Available from: http://www.mygrid.org.uk/usermanual1.7/webservice_dataproxy.html.
7. UniversityOfManchester. *Taverna Workflow Management System*. 15.07.2011]; Available from: http://www.taverna.org.uk/.
8. Barker, A., J.B. Weissman, and J.v. Hemert, *Orchestrating Data-Centric Workflows*, in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. 2008, IEEE Computer Society. p. 210-217.
9. Ludascher, B., et al., *Scientific workflow management and the Kepler system: Research Articles.* Concurr. Comput. : Pract. Exper., 2006. **18**(10): p. 1039-1065.
10. Barker, A., J.B. Weissman, and J.v. Hemert, *Eliminating the middleman: peer-to-peer dataflow*, in *Proceedings of the 17th international symposium on High performance distributed computing*. 2008, ACM: Boston, MA, USA. p. 55-64.
11. Weissman, J. and S. Ramakrishnan, *Using proxies to accelerate cloud applications*, in *Proceedings of the 2009 conference on Hot topics in cloud computing*. 2009, USENIX Association: San Diego, California.
12. Chafle, G.B., et al., *Decentralized orchestration of composite web services*, in *Proceedings of the 13th international World Wide Web conference on Alternate track papers \& posters*. 2004, ACM: New York, NY, USA. p. 134-143.
13. Kyprianou, N., *Hybrid Web Service Orchestration*, University of Edinburgh.
14. PlanetLab.org. *Planet Lab*. 10.07.2011]; Available from: http://www.planet-lab.org.
15. Wieland, M., et al., *Towards reference passing in web service and workflow-based applications*, in *Proceedings of the 13th IEEE international conference on Enterprise Distributed Object Computing*. 2009, IEEE Press: Auckland, New Zealand. p. 89-98.
16. W3Consortium. *XML Schema*. 20.06.2011]; Available from: http://www.w3.org/XML/Schema.
17. W3Consortium. *Web Services Description Language*. 22.06.2011]; Available from: http://www.w3.org/TR/wsdl.
18. W3Consortium. *Simple Object Access Protocol*. 26.06.2011]; Available from: http://www.w3.org/TR/soap/.
19. ApacheSoftwareFoundation. *Apache Tomcat*. 10.06.2011]; Available from: http://tomcat.apache.org/.
20. SQLite.org. *SQLite*. 25.06.2011]; Available from: http://www.sqlite.org/.
21. Oracle. *GlassFish*. Available from: http://glassfish.java.net/.
22. MSDN. *Model-View-Controller*. 25.08.2011]; Available from: http://msdn.microsoft.com/en-us/library/ff649643.aspx.
23. PetalsLink. *EasyWSDL*. 28.08.2011]; Available from: http://easywsdl.ow2.org/.
24. Oracle. *METRO*. 20.06.2011]; Available from: http://www.oracle.com/technetwork/java/index-jsp-137004.html.

25. Fowler, M., *Patterns of enterprise application architecture*. The Addison-Wesley signature series. 2003, Boston: Addison-Wesley. xxiv, 533 p.
26. MovableType. *Haversine*. 28.08.2011]; Available from: http://www.movable-type.co.uk/scripts/latlong.html.
27. Quova. *Quova IP geolocation*. Available from: http://api.quova.com/.
28. FreeGeoIP. *Free Geo IP geolocation*. 15.08.2011]; Available from: http://freegeoip.appspot.com/.
29. ShowMyIP. *Show My IP geolocation*. 14.08.2011]; Available from: http://www.showmyip.com/.