Distributed Programming in Cloud Computing Platforms

Enes Uysal

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Carlos Martins Delgado

Examination Committee
Chairperson: Prof. José Carlos Alves Pereira Monteiro
Supervisor: Prof. José Carlos Martins Delgado
Member of the Committee: Prof. Rui António dos Santos Cruz

May 2016
Resumo

Em ambiente de computação em nuvem, as soluções de programação distribuída que contem interoperabilidade entre sistemas heterogêneos são essencialmente baseadas em tecnologia SOA ou REST, com base em HTTP/1.1, XML e JSON. O problema é que estas tecnologias foram desenvolvidas no contexto Web, em que o principal objetivo é a integração de sistemas existentes e não a programação distribuída, com eficiência e desempenho como preocupações de topo. Como conciliar estas duas visões, em ambiente global de computação em nuvem, é o tópico principal desta dissertação.

SOA e Web Services são tecnologias pesadas e complexas. A popularidade do REST deve-se mais à sua simplicidade do que ao seu mapeamento no paradigma dos serviços. No entanto, qualquer delas é pouco eficiente porque eles são baseados no HTTP/1.1 e em linguagens de descrição de dados baseadas em texto (XML e JSON). O resultado é que a descrição de Web Services (WSDL) e linguagens de programação de alto nível ficam ineficientes e complexas. A versão binária do XML não é mais do que uma compressão de dados apenas para efeito de transmissão (comprimida na emissão e descomprimida na receção). A solução a usar neste trabalho usa formato binário de dados de forma nativa, não necessita de schemas e o ambiente de execução usa WebSockets, muito mais eficientes do que HTTP/1.1, embora mantendo alguma compatibilidade.

Uma nova solução foi desenvolvida para programação distribuída em ambiente de computação em nuvem para desenvolver aplicações distribuídas. Em comparação com uma solução clássica, que é baseado em SOA e REST, esta solução utiliza interoperabilidade assimétrica, em que o emissor e recetor podem ter tipos de dados diferentes, desde que o emissor satisfaça a parte do tipo de dados do recetor que realmente usa. O objetivo deste trabalho é explicar esta solução e seu ambiente de execução, comparando-a com as soluções correspondentes com tecnologias baseadas em SOA (Web Services) e REST.

Palavras-chave: SOA, REST, XML, JSON, HTTP, WebSocket
Abstract

In cloud environments, distributed programming solutions that cover interoperability between heterogeneous systems are essentially based on the SOA or REST architectural styles, which in turn are based on HTTP/1.1, XML and JSON. The main problem is that these technologies were developed in a context in which the main objective is the high-level integration of existing systems and not the programming of distributed systems, with efficiency and performance as top concerns. How to reconcile these views in a global distributed environment, such as a cloud computing platform, is the main topic addressed by this thesis.

SOA and Web services are heavy and complex technologies. The popularity of REST is more due to its simplicity than due to its mapping to the paradigm of services. Both exhibit a higher degree of coupling than applications require, since both interacting parties share the same data schema and therefore need to cater for all the features of that schema, even if some of them are never used at all. There is also an inefficiency problem, since these technologies are based on HTTP/1.1 and text-based data description languages (XML and JSON). The binary version of XML is no more than a compression mechanism, for transmission purposes only (compressed and decompressed at the endpoints).

This dissertation proposes a new solution to support interoperability of distributed programming systems in cloud computing environments. Contrasting with the schema sharing of SOA and REST, this thesis proposes asymmetric interoperability, in which the sender and receiver may have different schemas, as long as the schema of the sender complies with (satisfies) the part of the schema of the receiver that it actually uses. The aim of this work is to explain this solution and its execution environment, and also to compare it with the corresponding solution technologies which are based on SOA (Web Services) and REST.

Keywords: SOA, REST, XML, JSON, HTTP, WebSocket
Contents

Resumo ........................................................................ iii
Abstract ........................................................................ v
List of Tables ................................................................... ix
List of Figures ................................................................... xi
Glossary ......................................................................... xiii

1 Introduction ................................................................. 1
  1.1 Overview ............................................................... 1
  1.2 Context ................................................................. 1
  1.3 The problem ......................................................... 2
  1.4 The solution ......................................................... 4
  1.5 Contributions ....................................................... 5
  1.6 Organization ......................................................... 5

2 State-of-the-art ......................................................... 7
  2.1 Overview ............................................................... 7
  2.2 Web Services ........................................................ 7
  2.3 SOAP Web Services ................................................ 9
  2.4 REST Web Services ................................................ 11
  2.5 Serialization formats (XML, JSON and Binary) ............... 13
  2.6 HTTP/2 ................................................................. 14
  2.7 Web Sockets ........................................................ 15

3 Interoperability ......................................................... 17
  3.1 Overview ............................................................... 17
  3.2 Service Interoperability ........................................... 17
  3.3 Data Interoperability ............................................... 18

4 Architecture of the solution ....................................... 27
  4.1 Overview ............................................................... 27
  4.2 Asymmetric interoperability ..................................... 27
  4.3 Binary Format ........................................................ 33
5 Implementation
  5.1 Overview .......................... 37
  5.2 Binary Level Serialization .......................... 37
  5.3 Receiver and Compliance .......................... 40
  5.4 Message Transportation .......................... 46
  5.5 Deployment .......................... 48

6 Comparison with existing technologies .......................... 51
  6.1 Overview .......................... 51
  6.2 Qualitatively and Quantatively Comparison .......................... 51
    6.2.1 Textual or binary representation .......................... 51
    6.2.2 Message protocol .......................... 52
    6.2.3 The interoperability problem .......................... 52
    6.2.4 Full interoperability in SOA and REST .......................... 53
    6.2.5 Asymmetric interoperability .......................... 54

7 Conclusions .......................... 57
  7.1 Achievements .......................... 57
  7.2 Future Work .......................... 58

Bibliography .......................... 59
List of Tables

2.1 URL Example .................................................. 12
2.2 HTTP Methods ................................................. 12
2.3 HTTP/2 and WebSocket ...................................... 16

4.1 TLV format (Type-length-value) ............................. 34

5.1 .Net Serialization Person Object ............................ 38
5.2 Java Serialization Person Object .............................. 38
5.3 Average sizes (in bytes) of several resource representations. ............................................. 40
5.4 General properties of the consumers for experiment. .......................................................... 41
5.5 General properties of the providers for experiment. .......................................................... 41
5.6 Information about input parameter of operations in the provider for experiment. ............ 43
5.7 Compliance and Conformance between the Query message and Weather objects ................. 46
5.8 Compliance and Conformance between the Cloud Servers and Mobile phones. .................. 46
5.9 Average times (in ms) of HTTP-based solutions and Web Socket ........................................... 47
# List of Figures

1.1 SOAP and REST Approach. ......................................................... 2

2.1 Simple Web Service Architecture. ............................................. 7
2.2 WSDL (Web Services Description Language). ............................... 10
2.3 SOAP (Simple Object Access Protocol). ....................................... 10
2.4 REST URLs and HTTP methods. ................................................ 12

3.1 Interoperability in SOA and REST. .............................................. 18
3.2 Step 1 to generate client code from WSDL file. ............................... 25
3.3 Step 2 to generate client code from WSDL file. ............................... 25

4.1 Asymmetric interoperability. ..................................................... 28
4.2 Partial interoperability, based on compliance and conformance. .......... 29
4.3 The result of Scenario 1,3 and 4 execution. ................................. 31
4.4 The result of Scenario 2 execution. ............................................. 31
4.5 Binary representation of primitive resource. .................................. 35

5.1 Average Execution times for serialization of XML, JSON and Binary .... 40
5.2 Message Transportation with Web Sockets. .................................. 47
5.3 Average times (in ms) of HTTP-based solutions and Web Socket. ........ 47
5.4 The characteristics of Java application server. ............................... 48
5.5 The characteristics of .Net application server. ............................... 49

6.1 Data interoperability. ............................................................ 54
6.2 Asymmetric interoperability. .................................................... 55
Glossary

10gen 10gen is an American software company that develops and provides commercial support for the open source database MongoDB, a NoSQL database that stores data in JSON-like documents with flexible schemas.

API In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software and applications.

Android Android is a mobile operating system (OS) currently developed by Google, based on the Linux kernel and designed primarily for touch-screen mobile devices such as smartphones and tablets.

BSON BSON is a computer data interchange format used mainly as a data storage and network transfer format in the MongoDB database.

CSS Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language.

DOM Document Object Model (DOM) is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML, and XML documents. The nodes of every document are organized in a tree structure, called the DOM tree. Objects in the DOM tree may be addressed and manipulated by using methods on the objects.

EXI Efficient XML Interchange (EXI) is a binary XML format.
<table>
<thead>
<tr>
<th>FTP</th>
<th>The File Transfer Protocol (FTP) is a standard network protocol used to transfer computer files between a client and server on a computer network.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip</td>
<td>Gzip is a file format and a software application used for file compression and decompression.</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language, commonly abbreviated as HTML, is the standard markup language used to create web pages.</td>
</tr>
<tr>
<td>HTTP/2</td>
<td>HTTP/2 (originally named HTTP/2.0) is the second major version of the HTTP network protocol used by the World Wide Web.</td>
</tr>
<tr>
<td>HTTP</td>
<td>The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.</td>
</tr>
<tr>
<td>Interoperability</td>
<td>Interoperability is a characteristic of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, in either implementation or access, without any restrictions.</td>
</tr>
<tr>
<td>JAR</td>
<td>JAR (Java Archive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation (JSON) is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.</td>
</tr>
<tr>
<td>MongoDB</td>
<td>MongoDB is a free and open-source cross-platform document-oriented database.</td>
</tr>
<tr>
<td>NoSQL</td>
<td>A NoSQL database provides a mechanism for storage and retrieval of data which is modeled in means other than the tabular relations used in relational databases.</td>
</tr>
</tbody>
</table>
REST  Representational state transfer (REST) is a programming architectural implementation intended to increase the efficiency of communication in computing systems.

RSS  RSS (Really Simple Syndication) uses a family of standard web feed formats to publish frequently updated information: blog entries, news headlines, audio, video.

SAX  SAX (Simple API for XML) is an event-driven online algorithm for parsing XML documents.

SMTP  Simple Mail Transfer Protocol (SMTP) is an Internet standard for electronic mail (email) transmission.

SOAP  SOAP, originally an acronym for Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on application layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOA  A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network.

TCP  The Transmission Control Protocol (TCP) is a core protocol of the Internet protocol suite.

URL  A Uniform Resource Locator (URL), commonly informally termed a web address is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it.
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WSDL</strong></td>
<td>The Web Services Description Language (WSDL) is an XML-based interface definition language that is used for describing the functionality offered by a web service.</td>
</tr>
<tr>
<td><strong>WebSocket</strong></td>
<td>WebSocket is a protocol providing full-duplex communication channels over a single TCP connection.</td>
</tr>
<tr>
<td><strong>XML</strong></td>
<td>Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.</td>
</tr>
<tr>
<td><strong>iPhone</strong></td>
<td>iPhone is a line of smartphones designed and marketed by Apple Inc. They run Apple’s iOS mobile operating system.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Overview

This chapter gives information about the subject of the thesis, the area of interest in that subject and also general information about current solutions.

1.2 Context

Distributed applications are required in most of the central application sectors, including e-commerce, e-banking, e-learning, e-health, telecommunication and transportation[1]. This implies that the Internet plays an important role in business, administration and our everyday activities. Distributed applications that share information do not need to be built with the same technologies. The fundamental problem is the programming of distributed applications with all the basic interoperability problems. They involve distributed platforms and heterogeneous components.

On the other hand, cloud computing platforms create new challenges for distributed systems. They bring new opportunities such as scalability, dynamic instantiation, location independence, application management and multi-tenancy (several users using the same application independently)[2]. However, they also create distributed platforms, so they need to be able to support distributed applications as easily as possible. That shows the importance of interoperability in cloud environments, because different cloud providers can have different properties, and they need to communicate with each other.

The subject of the thesis is to research current solutions of distributed applications and their implementation in cloud computing environment and propose a design of the alternative solution that is both simpler and more effective than existing ones.

We take an Android mobile phone as an example, since nowadays a huge part of the population use smartphones that check news, weathercast reports and other information over the Internet. Let us say there is an application inside that Android phone which informs you with the present weathercast report of your city. That application can be written in Java, for example. The application gets the current weathercast report over the Internet by using weathercast providers, so basically that Java application
sends queries to the weathercast providers over the Internet. When the response arrives from the weathercast provider, the Java application parses received data and it displays information to the mobile phone screen. During the data exchange that occurs, both mobile phone and weathercast provider must understand each other, even if they don’t have execution code that is written in the same language, because the provider could be working in a cloud provider and written in C# language, for instance. The data types used in C# and Java are different, so they will not understand each other naturally. This interoperability (how to interconnect different programs written in different languages, running on different platforms) issue can be solved with current integration technologies such as SOA or REST using XML or JSON over HTTP and both platforms can communicate between each other. Finally, the last weathercast information can be obtained from the Android phone or iPhone using the same weathercast provider even they were developed with totally different technologies.

1.3 The problem

Distributed applications need to interact among themselves. In this respect, they need to solve the interoperability problem. Distributed applications need to understand each other to achieve meaningful and useful collaboration. The traditional integration technologies are based on either SOA or REST, as seen in Figure 1.1. They use the document concept as the foundation, with a data description language as the representation format. They use schema sharing as the interoperability mechanism (both sides use the same schema, such as XML Schema) or using previously known data types. Other factors, such as a connectionless protocol (HTTP/1.1) and XML, JSON (and by extension SOA and REST) that do not support binary in its main features, because the solutions were also based on text (XML or JSON) and contextual information, imply limitations for many applications.

Figure 1.1: SOAP and REST Approach.
The main problem of the traditional integration technologies is coupling between the provider and the consumer, because both customer and provider are forced to implement full interoperability (for example, sharing an XML schema). This leads to a greater coupling than needed.

While solving data interoperability, the problem of using XML or JSON is that XML and JSON are inefficient in computer terms due to parsing data to build a DOM tree and checking schemas to validate the structure of the message.

The problem can be explained by an example, as described in the Android mobile example in section 1.2 (Context), in which you can have two different technologies and they can communicate with each other. If they use the SOA architecture style (namely, SOAP and Web Services) you need to use XML, which both interacting parties can understand, since it is a common language. However, you still have some steps to deal with, such as a stub, which is used by a client application to access a remote service. The stub looks like the service interface[3]. It generates an appropriate XML message and sends the message over HTTP, then you have an XML message that can be read by your mobile phone or weathercast provider, and the XML message is parsed to build a DOM tree to be used. That is done by sharing the same schema to know the structure of the message. There are some problems with sharing a schema because it brings coupling problems. When you want to see the weathercast report on your phone, firstly, your phone creates a query message in XML with a stub generator then uses HTTP for message transportation. The created message should be validated by the schema because if it doesn’t obey the schema rules, the provider cannot understand the message. When the message arrives to the provider, it validates message regarding the same schema and parses that XML message so that its own language objects can understand the message. After understanding the request, it provides a response regarding the query message. There are some problems with using XML schema because your phone application or weathercast provider cannot alter this schema without informing each other. If one of them alters the schema without informing the other, then the other part cannot understand the message. That creates coupling between the mobile application and the weathercast provider. After message creation by weathercast provider, this response message again needs to be converted to XML message with a stub generator, then the provider sends that response message to your phone. Now, your phone uses XML schema and XML parsers to understand the XML message. Finally, it uses these data to display information to you. As seen in that example, it is not very easy to communicate between systems that use totally different technologies.

In the case of REST, you can use different message format instead of XML, for example, JSON, which is easier to parse and lighter than XML, but in REST, both languages need to know the message structure so that they can then parse and understand the message. If your weathercast application uses REST to communicate with weathercast providers, then you don’t need to create an XML message to send to the weathercast provider. You can use other formats to send the message to a server. Assume that you use JSON and your mobile phone creates a query in JSON format, then it needs to know unique URL of the weathercast provider and HTTP verbs (GET, POST, PUT, and DELETE) because in REST, different HTTP methods can be used with the same URL. When a message arrives to the weathercast providers, this message will be parsed with DOM parsers to understand the query. In
that case, your phone and weathercast provider must know the structure of the JSON message, so that they can both understand each other. REST may seem more flexible, in the sense that, if the server changes the links then the client will follow this change automatically by using the new links. However, since the client must understand the structure of the responses, REST imposes the constraint of returning representations using standardized or pre-agreed media types to achieve this. Your mobile phone application or weathercast provider cannot alter this structure of JSON without informing each other. After the message parsed by the provider, it creates a response message in JSON format to send back your phone and now your phone displays the weathercast to you.

Current integration technologies, based on either SOA or REST, use a data description language as the representation format and schema sharing as the interoperability mechanism. Other factors, using a connectionless protocol (HTTP/1.1) and the lack of native support for binary data and contextual information limit for many applications, although they are not imperative.

The main problems with using traditional integration technologies can be summarized as follows:

- Data interoperability problems based on XML and JSON (Stub generation, DOM parsing)
- Service interoperability problems based on SOA and REST (Coupling with schema sharing)
- The underlying protocol, usually HTTP/1.1, without binary support.

These are the main problems with using traditional integration technologies. In section 1.4, an alternative solution will be proposed to solve described problems.

1.4 The solution

As a solution to the problems mentioned above (in Section 1.3), such creating coupling between the provider and the consumer, we propose a new solution that provides the maximum decoupling possible, while ensuring the minimum interoperability requirements and allowing the client or the sender to change their schema, as long as the actual used parts do not change. Current solutions use schema sharing to understand message structure. Schema sharing creates unnecessary coupling. The new solution introduces compliance (the consumer must satisfy the requirements established by the provider to accept requests sent to it)[4] and conformance (the provider must fulfill the expectations of the consumer regarding the effects of a request)[5], instead of sharing the same schema. Basing interoperability on compliance and conformance avoids the problem of having to define schemas as separate documents and to agree upon them beforehand. As long as compliance and conformance hold, any two resources can interoperate, even if they change their structure without informing each other. The new solution uses binary directly for message transportation, instead of using XML or JSON. XML and JSON are based on text, which is heavy to parse. Using binary reduces complexity and improves performance. For the performance of message transportation, the new solution uses Web Sockets and HTTP/2 protocol (a binary protocol with small message frames) instead of using the classical HTTP/1.1. Chapter 3 and Chapter 4 give more details and examples of the topics that are mentioned in the current section.
1.5 Contributions

In this thesis, the aim is to show that it is more than just an implementation and the value of this thesis lies in the demonstration of conclusions, with respect to Web Services and REST. The contributions are made in thesis as follows:

- Implementing a new solution for interoperability problem with compliance[4] and comformance[5], which are not known very well or widely used on commercial tools.
- The solution allows using the best features of SOA and REST at the same time.
- The solution will be demonstrated through experiments. It will be shown with results that it is a better solution for application interoperability, and also that it is easier to implement. Furthermore, it will be assessed for performance to show the advantage of using binary, by comparison with current text-based solutions (XML, JSON).

1.6 Organization

The organization of thesis is structured as follows:

- State of Art - Chapter 2 details some aspects of existing solutions for distributed systems for a cloud environment, namely SOA, REST, text-based data (XML and JSON). It also details recent technologies that will be used, namely HTTP/2 and Web Sockets.
- Interoperability - Chapter 3 describes the interoperability problem and explains the differences in perspective from classical solutions to our new solution.
- Architecture of the solution - Chapter 4 provides some insight on how our system can be used and gives a general overview of how and why it works.
- Implementation - Chapter 5 goes into more depth regarding the inner workings of our system and describes our implementation.
- Comparison with existing technologies - Chapter 6 presents the benchmarks used to evaluate our system with current solutions and the results obtained.
- Conclusions - Chapter 7 summarizes the work described in this thesis, the results achieved, and what are its main contributions. It also presents some possible future improvements to our proposed solution.
Chapter 2

State-of-the-art

2.1 Overview

A detailed description is provided throughout this section concerning the state-of-the-art technologies existing nowadays in the market. The main integration technologies available today are based on Web technologies, namely HTTP, XML, JSON, Web Services using either SOA or REST architectural styles. This section also gives information about new technologies such as HTTP/2 and WebSocket technologies that will be used in the implemented solution.

2.2 Web Services

Web services allow two different machines or two different pieces of code to communicate to each other. Two different applications can communicate to each other over the network. They can call methods of each other over the network by using web service technology (Figure 2.1). The other advantage of using web services is that it actually is a standard technology, because it is not really specific to Java programming language or any other programming language. You can develop web services using all other programming language technologies. For example, you can develop a web service with the programming languages like Java, .Net, Python, C++ or others. The advantage of the web service standards is that a product or system, whose interfaces are completely understood, can interoperate with other products or systems, present or future[6].

Figure 2.1: Simple Web Service Architecture.
For example, let us say that you have written a web service in Java programming language and let us say that you have another web service that is written in .Net programming language. The Java web service can call the .Net web service and the .Net web service can also call the Java web service. One web server can call other web servers, but it just doesn’t have to be a web service because it can be a client application. Let us say that you have a client application written in C++, so that the C++ app calls the Java web service or .Net web service. You can actually have different applications written in different technologies that communicate with each other at execution time, and they can call each other. For example, you can have a set of business web services that are implemented in .Net and your Java application can use them.

Web Services are exposed to the Internet for programmatic access. They are online APIs (application programming interface) that you can call from your code. When you want to call any API that is written by someone else from your Java code, you basically add a JAR (Java Archive) or classes to your class path, and executions are done inside of the machine or single environment. In the case of web services, however, you have different pieces of code deployed over different machines, and they call methods of each other over the network. For example, you must have seen different apps or games that can post to your Facebook wall even if these games are not designed by Facebook. Basically, they do this by calling online APIs. Companies like Facebook or Twitter publish web services that let other developers call them from their code, so other application developers can actually write code to consume these services, and they can post things on Facebook or Twitter. They can read or access data from Facebook or Twitter using the APIs of the web services that Facebook or Twitter have provided.

Web services send a request to a provider server and get the response. For example, Twitter has web site URL “http://www.twitter.com”. When you access this URL on your browser, you get an HTML response that lets you read and write tweets. They have HTML elements for data and also CSS files for styling. This is because web pages that you see are made for human conception. Developers know that there is actually a human behind of browser that is reading these tweets, so they want to make sure that it is formatted properly, so that it is easy to access and to read. Twitter has also another URL, “http://api.twitter.com”, which does a lot of the same things as “http://www.twitter.com” does, but it behaves a bit differently, for instance this API gives you a response which doesn’t have HTML or CSS code. It contains data, but it is XML or JSON format and there are specific URLs for different operations. This is what the developers can use from their code to read or write to Twitter, so this data is actually very easy to parse, to convert and to use in their code. In this case, there is no need to have HTML and CSS files.

There are primarily two different types of web services. One of these types is called Simple Object Access Protocol (SOAP) web service, and another type called as a REpresentational State Transfer(REST) web service. SOAP is older of these two and REST is the newer entry to the web service world, but both are used popularly. The next section focuses on SOAP and REST web services.
2.3 SOAP Web Services

Simple Object Access Protocol, or SOAP as it was the first attempt to standardize a web service interface. It is based on sending an XML message to a service in a specific XML format, and receiving an XML response in another specific format. The message can be sent across different transports, including HTTP, FTP (File transfer Protocol), SMTP (Simple Mail Transfer Protocol) and more[7]. The specification does not dictate the transport over which the message should be sent, but most implementations send the XML message over HTTP.

Now SOAP web service will be explained shortly by an example. Let us start with an example of a Java application. Let us say you have implementation class, and you want to share this implementation class with other developer projects. How would you share this with a consumer class? The best way to share this implementation class would be to contract it with a Java interface and other consumers would consume this class through this Java interface. They would call the implementation through the interface, so they get a contract. They get the methods, arguments and written types through the interface. They actually call the methods of the implementation class, so how does this work in case of a web service? Let us say that you have a web service implementation, and you want to share details of this web service to consumers. Does it work with an interface? Probably it will not work because as discussed before you don’t know what technology is consuming it. It might be a .Net application or a C++ application, so if you have a Java web service you might want to give some kind of information that its consumer respects that technology, and that it can actually consume the service. Let us say that the consumer is .Net and if you give a Java interface class to a .Net application it will not work because they are different technologies, so the technology that you are going to share with the web service consumer has to be technology independent. It should be something that any application or any technology can understand. Creators of SOAP discussed about a format understandable by all technology with all consumers and decided on XML. In case of a web service, you share that contract as an XML document. This XML document is actually called WSDL(Web Services Description Language))[8].

A WSDL (a set of conventions on XML usage) document contains the contract to web service and when you create the web service, you share the WSDL document of that web service to the consumers (Figure 2.2). The WSDL document can be produced manually but there are tools that generate WSDL for the web service. There is a stub generator for every language to create that WSDL document. It is something that you need to share with consumers, and it is an XML document, so it is readable by any application in any language, such as .Net, C++ or Java. Applications can parse this XML[9] and obtain information about the service. This WSDL document is similar to an interface. It has operations, arguments and types to return, so that consumer applications will have an idea what to call and how to call.

The next question is how this exchange happens and how do you actually send this information. Let us say that you have a method whose input argument is a string. A string in Java is obviously different from string in .Net. How do you exchange this between client app and web service? When you exchange information about the input argument or return types, you need to exchange it in a format that
all different technologies can understand what you are passing, and the system should be able to send a return type back in a language that all interacting partners can understand. Again, this format is XML. When you send any information across the network from a client to a web service, and the return type back to the client, the data has to be in XML format. You are not really sending a Java string. There is a specification about how you need to send all these different input and output arguments. Basically, any type needs to be send in a specific XML format, which is a protocol that both sender and receiver use, called SOAP (Simple Object Access Protocol). Using this protocol, different technologies can access objects and data (see Figure 2.3). It is supposedly simple, hence the name. All different technologies written in different languages can understand what they all taking about with this protocol.

Figure 2.2: WSDL (Web Services Description Language).

Figure 2.3: SOAP (Simple Object Access Protocol).
Now you know what is the mechanism, you know what needs to be sent, and you know how to send. The mechanism uses the SOAP protocol, but who does the conversion? For example, you have your string object or complex object, so how do you convert it from a Java object to a SOAP message? The conversion is actually done with an intermediate class, which takes care of converting all your objects into a SOAP message. The method calls are actually done by SEI (Service Endpoint Interface)[10]. The SEI accesses the interface of your web service endpoint. You have an interface at your client application to the service endpoint, which translates all web service calls to SOAP messages, and then it makes sure that the other things are able to understand this message. You don’t have to write this class and all the conversion because it is automatically generated for you. When you make a web service call you don’t worry about where the web service is. All you need to do is to have this endpoint interface. It is specific to the technology in which you are developing your programs. When you have a Java application, you will have a specific SEI for Java application. That specific SEI knows how to convert Java objects to SOAP messages. Let's say your .Net application calls the same web service, so you will have SEI for .Net that knows how to convert .Net objects to SOAP messages.

2.4 REST Web Services

REST (Representational State Transfer) was created in 2000 by Roy Fielding[11]. Developed in an academic environment, this protocol embraces the philosophy of the open Web. Instead of using XML to make a request, REST relies on a simple URL in many cases. In some situations, you must provide additional information in special ways, but most Web services using REST rely exclusively on obtaining the needed information by using the URL approach. REST can use four different HTTP verbs (GET, POST, PUT, and DELETE) to perform tasks.

Resources are the fundamental building blocks of web-based systems. A resource is anything to be exposed to the Web, from a document or video clip to a business processor device. A URL uniquely identifies a web resource, and at the same time makes it addressable, or capable of being manipulated using an application protocol such as HTTP. The relationship between URLs and resources is many-to-one[12]. A URL identifies only one resource, but a resource can have more than one URL. That is, a resource can be identified in more than one way, much as humans can have multiple email addresses or telephone numbers. This fits your need to identify real-world resources in more than one way. Everything in REST web services has a URL, unique and standard. For example, Facebook, when you open an account on Facebook, you will get a profile page that obviously is dynamically generated, so that whenever there is a new profile, it is basically the same page which does the same processing but renders different content depending on the profile that you are watching. In REST web services, you need to think of resources and to create unique URLs for them. For example, you are creating a weathercast application, and you want to get weather information for different cities of Portugal, so your URL needs to be unique for each city as seen in Table 2.1.

In REST web services, different actions can be done with the same unique URL. For example, if you are an administrator of a weathercast application, you want to get data of Lisbon city or if you want to
update data of Lisbon city or deleting data of Lisbon city, you can use the same unique URL for all these methods by using different HTTP verbs. This can be implemented with only one unique concrete URL and four HTTP verbs. In fact, it's so compact that it can provide an overview in just a few lines, as shown in Table 2.2.

<table>
<thead>
<tr>
<th>Verb</th>
<th>URL or template</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/city</td>
<td>Create a new city</td>
</tr>
<tr>
<td>GET</td>
<td>/city/cityName</td>
<td>Request the data of city specified by the URL.</td>
</tr>
<tr>
<td>PUT</td>
<td>/city/cityName</td>
<td>Update an city data at the given URL with new information.</td>
</tr>
<tr>
<td>DELETE</td>
<td>/city/cityName</td>
<td>Logically remove the city identified by the given URL.</td>
</tr>
</tbody>
</table>

You have seen the requests so far, and understood what resource URLs and HTTP verbs are (Figure 2.4). You can make a request to resource URLs with HTTP verbs. Let us switch to responses now. After making a request, a RESTful web service will respond. The format of the response is important, because the client needs to handle that response. If that was a web application, the response would have been an HTML page with styling formatting. With a RESTful Web service, you don't need all this styling because the response can be basically XML or JSON. The following section (Section 2.5) gives detailed information about XML or JSON.
2.5 Serialization formats (XML, JSON and Binary)

The contents of messages in distributed systems need to be serialized to be sent over the channel with a format such as XML or JSON. A schema is used to transform the internal data structures into serial messages and vice-versa. The serialization formats used on the Web (e.g., XML, JSON) are text-based and thus verbose and costly in communications. Technologies have been developed to compress text-based documents, such as EXI (Efficient XML Interchange)[13], BSON[14] and others. However, these are compression technologies, which need text parsing after decompression.

Listing 2.1: Example Person Object Class

```java
public class Person {
    public String name;
    public int age;
    public String eyeColor;
}
Person p = new Person();
p.name = "Peter";
p.age = 34;
p.eyeColor = "blue";
```

Listing 2.2: XML presentation of Person Object Class

```xml
<Person>
    <name>Peter</name>
    <age>34</age>
    <eyeColor>blue</eyeColor>
</Person>
```

XML, which has retained the text markup style of HTML, has made computer-based clients easier, and it became the cornerstone of Web Services technologies, together with HTTP. This evolutionary transition is perfectly understandable in market and standardization terms, but it still constitutes a mismatch towards both humans and applications. XML is text based and support schemas. XML retains the look and feel of HTML, with text markup based on start and end tags as seen in the example (in Listing 2.2) of Person Object (in Listing 2.1). XML has generalized HTML by separating data from formatting and introducing a self-description with a schema, but retained much of its look and feel, still with a data document nature (just data, instead of a more complete service nature, by including code) and text with markup (lacks native binary support). There are technologies such as EXI, the binary XML format, which offer compression at less cost than using Gzip, and save processing, otherwise it is needed to decompress compressed XML[15].

JSON is much more popular than XML[16], as a simpler alternative to XML because it is more compact, especially if you have a large amount of data and also when the client is a browser which has a piece of JavaScript code running in the browser, so sending response data in JSON can be easily
parsed by the client. JSON delimits data with syntax tokens, with a simple grammar that bears some similarity with data structures in C-like languages as seen in the example (in Listing 2.3) of Person Object (in Listing 2.1). XML has targeted flexibility and generality, whereas JSON has emphasized simplicity, which is after all the secret of its popularity. There is also BSON, the binary JSON format devised by 10gen and used in their MongoDB NoSQL database, instead of JSON [17]. As described before, these are compression technologies, which need text parsing after decompression.

Listing 2.3: JSON presentation of Person object

```
{
  "Person": {
    "name": "Peter",
    "age": "34",
    "eyeColor": "blue"
  }
}
```

In spite of the differences, both suffer from drawbacks and limitations; They are text based, which means inefficient parsing and data traversal where all characters of a component need to be parsed to reach the next one, high memory consumption, relevant message transmission times and poor support for binary data. The serialization format should be as close as possible to the data structure to minimize the conversion effort in serialization and deserialization. Text can be thought as human readable and therefore, advantageous over binary, but this is true only for very simple documents, especially when using XML. Binary compression mechanisms exist [13][14], but this does not reduce the parsing time, since text needs to be recovered first. It would be better to follow the old example of programming languages, by using a source format for humans, a binary format for computers and a compiler to synchronize them[16].

### 2.6 HTTP/2

HTTP/2[18] is the second major version of the HTTP network protocol used by the World Wide Web. HTTP/2 will make our applications faster, simpler, and more powerful. The primary goals for HTTP/2 are to reduce latency by enabling full request and response multiplexing, to minimize protocol overhead via efficient compression of HTTP header fields, and to add support for request prioritization and server push[19].

The first important thing to notice about HTTP/2 is that it is not a replacement for the entire HTTP. The verbs, status codes and most of the headers will remain the same as today. HTTP/2 is about becoming more efficient in the way data is being transferred on the wire[18].

HTTP/2 breaks down the HTTP protocol communication into an exchange of binary-encoded frames, which are then mapped to messages that belong to a particular stream, and all of which are multiplexed within a single TCP connection. HTTP/2 communication is split into smaller messages and frames,
each of which is encoded in the binary format. This is the foundation that enables all other features and performance optimizations provided by the HTTP/2 protocol.

HTTP/2 Server Push allows servers to proactively send responses to client caches. In a typical HTTP workflow, the browser requests a page, the server sends the HTML in the response, and then needs to wait for the browser to parse the response and issue additional requests to fetch the additional embedded assets (JavaScript, CSS, etc.). Server push allows the server to speculatively start sending resources to the client. Here, the browser does not have to parse the HTML page and find out which are the other resources to load; instead, the server can start sending them immediately.

### 2.7 Web Sockets

Web Sockets\[20\] are a relatively new technology that promises to make websites more reactive by allowing lower latency interaction between users and the server. Web Sockets circumvent some of the limitations of HTTP, namely by adding binary support, and increases performance.

Web Sockets entail a protocol that supports communication between the client and the server/end-point using a single TCP connection. It sounds a bit like HTTP. The advantage that WebSocket has over HTTP is that the protocol is full-duplex (allows for simultaneous two-way communication) and it’s header is much smaller than the HTTP header, allowing for more efficient communication even with small packets of data. Web Sockets are also fundamental in the efficient support for binary data and increases performance. An example of WebSocket lifecycle is\[20\]:

1. A client sends the server a handshake request in the form of a HTTP upgrade header with data about the WebSocket it is trying to connect to.

2. The server responds to the request with another HTTP header, and this is the last time a HTTP header is used in the WebSocket connection. If the handshake was successful, the server sends a HTTP header telling the client it is switching to the WebSocket protocol.

3. Then the connection is opened and the client and server can send any number of messages to each other until the connection is closed. These messages have a small number of overhead bytes.

Comparing HTTP/2 against Web Sockets\[18\], as explained above they both support binary data and increase performance with respect to classical HTTP (HTTP/1.1). A lot of similarities between HTTP/2 and Web Sockets can be seen in Table 2.3:
Table 2.3: HTTP/2 and WebSocket

<table>
<thead>
<tr>
<th>Feature</th>
<th>HTTP/2</th>
<th>WebSocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headers</td>
<td>Compressed (HPACK)</td>
<td>None</td>
</tr>
<tr>
<td>Binary</td>
<td>Yes</td>
<td>Binary or Textual</td>
</tr>
<tr>
<td>Multiplexing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Prioritization</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Compression</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Direction</td>
<td>Client/Server + Server Push</td>
<td>Bidirectional</td>
</tr>
<tr>
<td>Full-duplex</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Chapter 3

Interoperability

3.1 Overview

Resources need to interact to accomplish collaboration, either designed or emergent. This necessarily entails some form of mutual knowledge and understanding, but this creates dependencies on other resources that may hamper resource evolution, and that is why interoperability is a necessary condition to achieve integration between different systems. Another important factor for integration between systems is decoupling, which says that resources need to be independent to evolve freely and dynamically. Unfortunately, independent resources do not understand each other and are not able to interact. Therefore, the fundamental problem of resource integration is to provide the maximum decoupling possible while ensuring the minimum interoperability requirements. While using data formats like XML and JSON, the systems need to work with schemas of XML or JSON documents, sent from the server or client sides, to understand the messages. Another problem of using XML or JSON to solve data interoperability is inefficiency in computer terms due to the need to parse text-based data, to build a DOM tree and to validate schemas to analyze the structure of the messages. This chapter gives detailed information about these interoperability issues.

3.2 Service Interoperability

Currently, enterprise integration is based on SOA with Web Services and REST with HTTP. These are two most used architectural styles for distributed interoperability. These styles use a symmetric arrangement in which a sender produces a message according to some schema, and the receiver uses the same schema to validate and to get the contents of the message, as seen in Figure 3.1. The message is sent over a channel between sender and receiver. In SOA with web services, the schema is usually expressed in XML Schema and WSDL. In the REST world, schemas are known as media types but perform the same role. The difference is that, instead of being declared in a separate document referenced by messages, they need to be previously known to the interaction resources, either by being standard or by having been previously agreed upon. In any case, the schema or media type must be the
same at both sender and receiver. This imposes coupling between the resources for all possible values that satisfy the schema, even if only a few are actually used. In either case, data types need to be fully known by both interacting resources.

![Diagram](image)

Figure 3.1: Interoperability in SOA and REST.

### 3.3 Data Interoperability

As described above (in Section 3.2), resources send requests to each other to invoke a given operation with a SOA or REST approach. These requests and their responses usually contain data, which is serialized, sent and reconstructed upon reception of the corresponding message. Most of the data types used in SOA and REST are expressed in XML and JSON. The previous chapter (in Section 2.5) gives detailed information about the advantages and disadvantages of XML and JSON. The sender and receiver need to interpret those data in a compatible way, which means they need a schema. These schemas can be in XML Schema and JSON Schema. JSON is a very popular serialization format, as a simpler alternative to XML. JSON Schema is currently just an IETF draft[21], but JSON Schema is raising interest as a simpler alternative to XML Schema. Both XML and JSON are text-based, which means inefficient parsing and all characters of a component need to be parsed to reach the next one. When using XML, there is some work with this document to understand the message, which is sent by the server or client sides. The solutions are Data binding, DOM (Document Object Model) or SAX (Simple API for XML)[22]. For example, XML data binding refers to a means of representing information in an XML document as a business object in computer memory. An XML data binder accomplishes this by automatically creating a mapping between elements of the XML schema of the document that needed to bind and members of a class to be represented in memory. Other solutions to create data or to access the data in the XML are DOM or SAX. They do same work in a different way. SAX provides a mechanism for reading data from an XML document that is an alternative to that provided by the Document Object Model (DOM). Where the DOM operates on the document as a whole, SAX parsers operate on each
piece of the XML document sequentially. In general, DOM is easier to use but has an overhead of parsing the entire XML before you can start using it.

Let us explain XML message creation with an example, referring to the example in Chapter 1 that demonstrates an interaction between an Android mobile phone and a .Net weathercast provider. In the case of the REST approach, an XML or JSON message is needed to be created to send to the server. Let us see how an XML message with Android phone application can be created to send for a weathercast provider. The code in Listing 3.2 explains how to create the XML message in Listing 3.1, by using Java.

**Listing 3.1: Example XML message to send weathercast provider**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<weather>
  <country>Portugal</country>
  <city>Lisbon</city>
  <date>30.03.2016</date>
  <weatherResult></weatherResult>
</weather>
```

**Listing 3.2: XML message creation with Java Code**

```java
DocumentBuilderFactory docFactory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
// root elements
Document doc = docBuilder.newDocument();
Element rootElement = doc.createElement("weather");
doc.appendChild(rootElement);
// country elements
Element country = doc.createElement("country");
rootElement.appendChild(country);
// city elements
Element city = doc.createElement("city");
city.appendChild(doc.createTextNode(cityInfo));
rootElement.appendChild(city);
// date elements
Element date = doc.createElement("date");
date.appendChild(doc.createTextNode(new Date().toString()));
rootElement.appendChild(date);
// weatherResult elements
Element nickname = doc.createElement("weatherResult");
nickname.appendChild(doc.createTextNode(""));
rootElement.appendChild(nickname);
TransformerFactory transformerFactory =
```

```java
```
19
TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
DOMSource source = new DOMSource(doc); // XML Message

After creation of the XML message with Java by using DOM, this message will be sent through a suitable message-level protocol (e.g., HTTP) and will be received by the provider. The weathercast provider will use XML data binding, DOM or SAX techniques to deserialize the message and to understand the query message. The code in Listing 3.3 explains how to parse an XML message using DOM in .Net.

Listing 3.3: XML message parsing with .Net Code

```csharp
string Country;
string City;
string date;
// Create an XmlReader
using (XmlReader reader = XmlReader.Create(new StringReader(xmlString)))
{
    reader.ReadToFollowing("weather");
    reader.ReadToFollowing("country");
    Country = reader.ReadElementContentAsString();
    reader.ReadToFollowing("city");
    City = reader.ReadElementContentAsString();
    reader.ReadToFollowing("date");
    date = reader.ReadElementContentAsString();
}
```

As seen in Listing 3.3, the receiver must know the format of the XML message. Otherwise, it cannot deserialize it correctly without knowing its structure. XML parsing code in the provider reads an XML document and uses DOM or SAX APIs to provide programmatic access to its contents and structure. After parsing the XML message, the .Net weathercast provider builds response data as an XML message and then sends it to your mobile phone. Your mobile phone needs to parse that XML response message using DOM or SAX API again to display the result on your phone screen. Here, in that example, it is assumed that both your mobile and weathercast provider know the same schema. The example schema for your mobile phone application and weathercast provider can be seen in Listing 3.4. They both first validate the message with a validator. For example, SAX API validator[23], as shown in Listing 3.5, otherwise the component cannot parse XML correctly and cannot communicate with each other.

Listing 3.4: Example schema for Weather XML file

```xml
<xs:schema attributeFormDefault=unqualified" elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="weather">
<xs:complexType>
<xs:sequence>
<xs:element type="xs:string" name="country"/>
```
<xs:element type="xs:string" name="city"/>
<xs:element type="xs:string" name="date"/>
<xs:element type="xs:string" name="weatherResult"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

Listing 3.5: Example XML message to send weathercast provider

```java
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(false);
factory.setNamespaceAware(true);
factory.setSchema(schemaFactory.newSchema(new Source[]{new StreamSource("weather.xsd")}));
SAXParser parser = factory.newSAXParser();
XMLReader reader = parser.getXMLReader();
// reader.setErrorHandler(new SimpleErrorHandler());
reader.parse(new InputSource("weather.xml"));
```

The same example can be implemented with a SOA approach. SOA leads to an architectural style that is an evolution of the RPC (Remote Procedure Call) style, by declaring and invoking operations in a standard and language independent way. Service description (WSDL document) must be obtained first by your mobile phone and weathercast provider, so that a contract can be established between that resource (the provider) and the resource that uses it (the consumer). A concrete schema, with the names used, must be specified and be compatible on both sides, otherwise a representation returned by a resource, for example, will not be able to be analyzed and understood.

A Web service (in Listing 3.6) was developed with .NET for weathercast provider. The following WSDL document (in Listing 3.7) will be created automatically for that Web Service by .NET.

Listing 3.6: SOAP Web Service with .Net

```csharp
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    
    [WebMethod]
    public Weather GetWeatherCast(Weather weather) {
```
weather.weatherResult =
    WeatherCastProvider.GetWeather(weather.country, weather.city,
    weather.date);

return weather;

So if your mobile application knows the WSDL document in Listing 3.7, it can easily call a method from weathercast provider by RPC (Remote Procedure Call) style to get the current weathercast info for the requested city, as shown in Listing 3.8.

Listing 3.7: WSDL Document for Weathercast Provider

```xml
<?xml version="1.0" encoding="utf-8"?>
<wSDL:definitions xmlns:tm="http://microsoft.com/wSDL/mime/textMatching/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wSDL/mime/"
    xmlns:tns="http://tempuri.org/"
    xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://schemas.xmlsoap.org/wSDL/soap12/"
    xmlns:http="http://schemas.xmlsoap.org/wSDL/http/"
    targetNamespace="http://tempuri.org/"
    xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
    <wSDL:types>
        <s:schema elementFormDefault="qualified"
            targetNamespace="http://tempuri.org/">
            <s:element name="GetWeatherCast">
                <s:complexType>
                    <s:sequence>
                        <s:element minOccurs="0" maxOccurs="1" name="weather"
                            type="tns:Weather" />
                    </s:sequence>
                </s:complexType>
            </s:element>
        </s:schema>
    </wSDL:types>
</wSDL:definitions>
```
type="s:string" />
</s:sequence>
</s:complexType>
<s:element name="GetWeatherCastResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="GetWeatherCastResult" type="tns:Weather" />
</s:sequence>
</s:complexType>
</s:element>
</s:schema>
</wsdl:types>
<wsdl:message name="GetWeatherCastSoapIn">
<wsdl:part name="parameters" element="tns:GetWeatherCast" />
</wsdl:message>
<wsdl:message name="GetWeatherCastSoapOut">
<wsdl:part name="parameters" element="tns:GetWeatherCastResponse" />
</wsdl:message>
<wsdl:portType name="WebServiceSoap">
<wsdl:operation name="GetWeatherCast">
<wsdl:input message="tns:GetWeatherCastSoapIn" />
<wsdl:output message="tns:GetWeatherCastSoapOut" />
</wsdl:operation>
</wsdl:portType>
</wsdl:binding name="WebServiceSoap" type="tns:WebServiceSoap">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="GetWeatherCast">
<soap:operation soapAction="http://tempuri.org/GetWeatherCast"
style="document" />
<wsdl:input>
<soap:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:binding name="WebServiceSoap12" type="tns:WebServiceSoap">
<soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="GetWeatherCast">
<soap12:operation soapAction="http://tempuri.org/GetWeatherCast"
To invoke a Web Service from any programming language, you need the stubs (interfaces) of the web service. From the client side, the business objects communicate with stub objects. The stub takes responsibility for the message and invokes the Web Service. Hence, the first step is to obtain a stub for Web Service. Developing client side code for a given WSDL is usually automatically generated for you by using a programming language. This is also called schema-first definitions that require a schema to be created prior to instances. When you are developing code in the client side, the following steps need to be completed to generate stubs code for a given WSDL file in Java.

- Click to “Add new file” as seen in Figure 3.2.
- Choose file type as “Web Service Client”.
- Choose WSDL URL and Copy and paste your WSDL URL address as seen in Figure 3.3.
- Click Finish.

After these steps, you can start to write client side code in Java, as shown in Listing 3.8.

Listing 3.8: SOAP Web Service Client with Java

```java
import CSharpWebService.Weather;
import CSharpWebService.WebService;
import CSharpWebService.WebServiceSoap;

public class Run {
    public Weather GetWeatherCast() {
```
Weather w = new Weather();
w.setCountry("Portugal");
w.setCity("Lisbon");
w.setDate(null);
WebService service = new WebService();
WebServiceSoap port = service.getWebServiceSoap12();
return (port.getWeatherCast(w));
}

Figure 3.2: Step 1 to generate client code from WSDL file.

Figure 3.3: Step 2 to generate client code from WSDL file.

As seen in Listing 3.7 for a very basic example, there is a big WSDL document in case of using SOA
architecture. The invocation is done by the client side to the stub, which receives the request message and passes the information on to service side business objects, at the service provider side (in Listing 3.6).

As seen with examples, a typical and pragmatic solution is to use Web Services and XML data with sharing schemas and namespaces, or RESTful APIs. RESTful APIs are simpler to use and require that schemas (media types) are standardized or pre-agreed. In those technologies, both customer and provider are forced to implement full interoperability (for example, by sharing an XML schema), even if only a fraction of the possible interactions is used. This leads to a greater coupling than needed. The proposed solution (in Chapter 4) reduces coupling by using partial interoperability, creating the maximum decoupling possible while ensuring the minimum interoperability requirements.
Chapter 4

Architecture of the solution

4.1 Overview

As discussed before, the idea is to explain the new solution and to compare it with current technologies (Web Services, REST, XML and JSON). This solution is a new programming technology, alternative to current solutions. Basically, this is an alternative to text-based data description (XML and JSON) technologies, for document sharing or service invocation between two completely different systems. The following sections will explain how to implement the interoperability, primitive data formats and the solution of binary-level compliance and conformance.

4.2 Asymmetric interoperability

In the new implemented solution, it will be shown that interaction is still possible with only a partial knowledge of types, as long as the characteristics actually used are included (partial interoperability). This is a way of getting closer to solving the fundamental integration problem, by reducing coupling to what is actually required. Coupling is bad, but no interaction is possible without coupling. The aim of the thesis is to minimize coupling as much as possible, down to the minimum level that ensures the level of interaction required by the resources to integrate.

In this solution, a different approach will be shown, based on compliance. Messages do not obey some external schema. Each message has one specific value. Messages are structured or primitive, with its own exclusive schema that is nothing more than a self-description, without the value variability that a type exhibits. This value and its description can be validated against an infinite number of schemas, those that have this particular value included in the set of their instances.

The receiver (in Figure 4.1) exposes a schema that defines the values it is willing to accept. When a message is received, its internal schema is checked against the receiver’s own schema. If it complies, which means it satisfies all the requirements of the receiver’s schema, the message can be accepted and processed. The advantage of this is that a resource can send a message to all the resources with schemas that the message complies with and, conversely, a resource can receive messages from any
resource that sends messages compliant with its receiving schema. In other words, coupling occurs only in the characteristics actually used by messages and not in all the characteristics of the schemas used to generate the message or to describe the service of the receiving resource. Since the schemas of the message and the schemas of the receiver are not agreed upon beforehand, they need to be checked structurally. Resources of primitive types have predefined compliance rules. Structured resources are compared by the names of components (regardless of order of declaration or appearance) and (recursively) by the compliance between structured resources with matching names. Since the order of appearance of named component resources may be different in the message and in the receiving schema, there is a need to map one onto the other. This is a form of polymorphism that increases the range of applicability of both sender and receiver, constituting a means to reduce coupling to only what is actually used. The sender and receiver no longer need to be designed for each other. As long as compliance is ensured, one resource can replace another. In this case, what is involved is conformance between the replacement and the resource replaced, stating that a former supports all the characteristics supported by the latter. When a resource can interact with another, although not entirely interoperable with it, this means that there is partial interoperability.

Figure 4.2 illustrates these concepts and differentiates compliance from conformance. It also describes complete transactions with both request and response. The interacting resources now perform the roles of consumer and provider with sender and receiver roles reversed from request to response.

Interoperability of a consumer with a provider is possible by satisfying the following properties:

Compliance[4], which means that the consumer must satisfy (comply with) the requirements established by the provider to accept requests sent to it, without which these cannot be honored.

Conformance[5], which means that the provider must fulfill the expectations of the consumer regarding the effects of a request (including eventual responses), therefore being able to take the form of (to conform to) whatever the consumer expects it to be.

In full interoperability, the consumer can use all the characteristics that the provider exposes. This is
what happens when schemas are shared. In partial interoperability, the consumer uses only a subset of those characteristics, which means that compliance and conformance needs only to be ensured for that subset. These properties are not commutative (e.g., if P complies with Q, Q does not necessarily comply with P), since the roles of consumer and provider are different and asymmetric by nature, but they are transitive (e.g., if P complies with Q and Q complies with R, then P complies with R).

In Figure 4.2, the resource A, in the role of consumer, has been designed for full interoperability with resource B, in the role of provider. A uses only the characteristics that B offers and B offers only the characteristics that A uses. Let’s assume that the provider of A is needed to change to resource X, which has been designed for full interoperability with resource Y, in the role of consumer. The problem is that A was designed to interact with provider B and X was designed to expect consumer Y. This means that, if you use resource X as a provider of A, B is how A views provider X, and Y is how X views consumer A. Ensuring that A is interoperable with X requires two conditions such as Compliance and Conformance.

For Compliance, B must comply with Y. Since A complies with B and Y complies with X, this means that A complies with X and, therefore, A can use X as if it were B, as it was designed for.

For Conformance, Y must conform to B. Since X conforms to Y and B conforms to A, this means that X conforms to A and, therefore, X can replace (take the form of) B without A noticing it.

Partial interoperability has been achieved by subsumption, with the set of characteristics that A uses as a subset of the set of characteristics offered by X. This inclusion relationship, without changing characteristics, is similar in nature to the inheritance-based polymorphism supported by many programming languages, but here it applies to a distributed context. It constitutes the basis for transitivity in compliance and conformance, as well as the mechanism to reduce coupling between two resources to the minimum required by the application.

Recall the example in Chapter 1 that demonstrates the interaction between an Android mobile phone and a .Net weathercast provider. Now let’s demonstrate how it works with the new implemented solution. Again, the mobile phone application creates a query and then the application sends it to the weathercast provider. After the message is created by the phone application, then the array of bytes of the message is transferred from the sender to the receiver, requiring a binary channel (Web Sockets or HTTP2). HTTP/1.1 can still be used, with BASE64 to overcome the limitation of HTTP/1.1 of not supporting
binary data. Everything is serialized to binary, then encoded in BASE64 and decoded at the receiver to recover the binary data. When the message arrives to the receiver (weathercast provider) then it tries to match against the type of the input parameter of each of its operations. If one is found, the message received is partially assigned to that argument and the operation invoked to create a response message. That response will be in binary format and send back to the mobile phone via a binary channel. An application will display that message to the user. Here again, the phone application is developed with Java programming language and the weathercast provider is developed with .Net technology. In the new solution, the mobile application will create a binary message and send it to the weathercast provider using Web Sockets or HTTP/2 technology. The example will be explained with four different scenarios:

Scenario 1:

Client side (the mobile phone application) has very simple Weather object class such as in Listing 4.1. This object class will be serialized to binary, and it will be sent to the server side (weathercast provider). The server side has Receiver class and Receiver class has different methods with AvailableMethod notation, and the formal argument of the methods with AvailableMethod notation will be used for mapping. One of the methods with AvailableMethod annotation has Weather1 object as an input parameter as shown in Listing 4.2. As seen (in Listing 4.3), that class has primitive values. Every field in that class has Mandatory notation and while the mapping is happening, type of primitive, name of primitive and value of primitive must be the matched.

Listing 4.1: Simple weather object class

```java
1 public class Weather
2 {
3     public String country = "Portugal";
4     public String city = "Lisbon";
5 }
```

Listing 4.2: AvailableMethod notation in Receiver Class

```java
1 [AvailableMethod]
2 public void MakeObjectC(Weather1 test)
3 {
4 }
```

Listing 4.3: Weather1 object in Receiver

```java
1 public class Weather1
2 {
3     [Mandatory]
4     public String country = "Portugal";
5     [Mandatory]
6     public String city = "Lisbon";
7 }
```
When a message (Weather object) is received by the server, it will be mapped to the Weather1 object as seen in Figure 4.3, because compliance has occurred. Then the server will run the operation, and the result will be sent to your mobile application.

Figure 4.3: The result of Scenario 1,3 and 4 execution.

Scenario 2:

In this scenario, everything is very similar to the previous one except city field has a different value in Weather class. The city field is Braga instead of Lisbon as seen in Listing 4.4. In this case, the result will not be the same as before. The city field is a mandatory field, then Weather object will not be mapped to Weather1 class as seen in Figure 4.4 and the server will not invoke any operation. The response will not be returned to the user’s phone application. However, in the future if the server adds one method to its Receiver class such as Weather3 class, as seen in Listing 4.6 with the city name Braga, then without changing anything in the code of your mobile application, a matching will occur between your mobile application and the weathercast provider, in which case the result will be returned.

Listing 4.4: Simple weather object class

```java
1 public class Weather {
2     3     public String country = "Portugal";
4     public String city = "Braga";
5 }
```

Figure 4.4: The result of Scenario 2 execution.

Scenario 3:

Now we have another class in Receiver class, which is called Weather2 as seen in Listing 4.5. Weather2 has one mandatory field and another one that is optional. Let us send again the same mes-
sage (in Listing 4.1) from the mobile phone application. In this case, there is a value for the country, but it is not mandatory. As expected, Weather object will be mapped to Weather2 object and the server will invoke an operation as seen in Figure 4.3, and a response will be returned to user’s phone application.

Listing 4.5: Weather2 object in Receiver

```java
public class Weather2
{
    public String country = "Portugal";
    [Mandatory]
    public String city = "Lisbon";
}
```

Listing 4.6: Weather3 object in Receiver

```java
public class Weather3
{
    public String country = "Portugal";
    [Mandatory]
    public String city = "Braga";
}
```

Listing 4.7: Simple weather object class

```java
public class Weather
{
    public String country = "Portugal";
    public String city = "Lisbon";
}
```

Scenario 4:

In this scenario, there are some small changes with respect to Scenario 3. There is no country name as seen in Listing 4.8, again using Weather2 class in the Receiver where the country is non-mandatory. In that situation, when the message is sent, it will match Weather2 because the country field is not a mandatory field and the server will use the default value instead of the one in the message. The result will be provided as seen in Figure 4.3 and the server will invoke an operation, and a response will be returned to the user’s phone application.

Listing 4.8: Simple weather object class for Scenario 4

```java
public class Weather
{
    public String country = "";
    public String city = "Lisbon";
}
```
As seen in the examples, there is a possibility to have an infinite number of classes in Receiver. If the message matches one of them, it runs the operation, returning eventually a result to the client. This eliminates data binding, stub generation and DOM inspection, because the receiver needs only to be concerned with the interface, which is defined at compile time. Mapping between the message and the formal argument of the receiver method is done in binary, automatically and based on compliance.

The idea is eliminating the need for static data binding (generation of stubs based on a schema). The Receiver has always a default value for the message, and only the message components that comply with what the receiver expects are assigned to the matching receiver argument's components. Matching is done by name. This means that the argument is either primitive (int, bool, string) or structured, with named components. Matching is either by type (primitive types) or name (if the argument is a structured type). Only the components that match are assigned to the formal argument of the operation.

As seen in the example, in the proposed implementation there is no need to generate an interface based on a schema or WSDL file. Messages do not obey some external schema. Each message has one specific value, which is structured or primitive with its own exclusive schema that is nothing more than a self-description, without the value variability that a type exhibits. This value and its description can be validated against an infinite number of schemas, those that have this particular value included in the set of their instances. Using this solution, we no longer need to produce a client stub, and either the client/producer or the server/reader can be changed without breaking interoperability, as longer as compliance (on the client side) and conformance (on the server side) are respected.

To make compliance possible, the formal argument of each operation must also be serialized, as seen in the example. This must also support optional components, which use the formal argument component if none in the message matches it. Therefore, the serialization methods in the static serialization class must include information on whether it is optional. The best is for each primitive data type to have two tags, one for mandatory and another for optional. Annotations allow us to define tags for primitive data types. Messages sent use only mandatory values. Serialized formal arguments can use both mandatory and optional.

### 4.3 Binary Format

One of the ideas in this new technology is using the binary format instead of using text or other formats, because binary is faster to write, communicate and read. When serialization or deserialization performance is compared with binary, XML and JSON, it is easily seen that binary format gives faster speed than the others, especially with large data[24].

On the other hand, using text is far more flexible. Textual representation entails parsing overheads. Messages received are parsed directly in binary, much faster than text parsing. The binary representation provides native support for binary data, has a smaller length and is faster to parse.

Defining a binary format which messages are serialized on send and recovered on reception. For binary format we use TLV (Tag, Length and Value) binary markup[25]. An array of bytes is used for each
resource serialized in a tag (a byte codifying each resource type), size, name (only on structure and resource components) and value (the actual sequence of bytes resulting from serializing the resource).

The binary format used to serialize the resources is TLV (Tag, Length and Value) binary markup, as shown in Table 4.1. This supports the direct integration of binary information but also facilitates parsing. The binary message format resulting from compilation of the source program uses self-description information only when needed. This allows maintaining all the information necessary to communicate in a standard and platform independent way.

<table>
<thead>
<tr>
<th>Format</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>A binary code, which indicates the kind of field that this part of the message represents.</td>
</tr>
<tr>
<td>Length</td>
<td>The size of the value field in bytes</td>
</tr>
<tr>
<td>Value</td>
<td>Variable-sized series of bytes which contains data for this part of the message.</td>
</tr>
</tbody>
</table>

As long as we control the serialization format, the serialization can be performed in one language and the deserialization in another. The binary format is always the result of serializing data in each language, with a tag, the number of bytes that follow and the serialized content. Recovering the serialized data is simply testing the tag to find the data type and then using the number of bytes and the serialized content.

Let us demonstrate this with an example, as seen in the example of Chapter 3 that explain the interaction between an Android mobile phone and a .Net weathercast provider. The phone has an application that is written in Java and sends the query message to a .Net weathercast provider, displaying the result regarding the respond from the .Net weathercast provider.

The procedure starts by serializing the query message to the binary format in Java. In this case, application in the phone will create the binary message instead of XML or JSON to send it to the weathercast provider, as shown in Listing 4.9.

```
public static void main(String[] args)
{
    Weather w = new Weather();
    w.country="Portugal";
    w.city="Lisbon";
    Message msg = new Message(w);
    byte[] msgToSend = msg.SerializeBinary();
    string response = clientEndPoint.sendMessage(msgToSend);
}
```

Creating a binary message will be similar to TLV (Tag, Length and Value) binary markup idea. Regarding the code in Listing 4.9, the provided binary will be as in Listing 4.10.
The weather object in Listing 4.9 is serialized by its primitive elements. So, Figure 4.5 shows how the first primitive element of the weather object is serialized to binary.

Listing 4.10: Creating serialized binary Message

```
[1, 3, 0, 0, 0, 7, 0, 0, 0, 8, 0, 0, 0, 8, 99, 111, 117, 110, 116, 114, 121, 80, 111, 114, 116, 117, 103, 97, 108, 1, 3, 0, 0, 0, 4, 0, 0, 0, 4, 0, 0, 0, 6, 99, 105, 116, 121, 76, 105, 115, 98, 111, 110, 1, 3, 0, 0, 0, 4, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 100, 97, 116, 101]
```

Figure 4.5: Binary representation of primitive resource.

The first part of binary ([1]) in Figure 4.5 tells that serialization starts by checking if the object is primitive or structured. After that step, the next part ([3]) informs the type of the primitive object. In that case it is a String object, and then the next part of binary ([0, 0, 0, 7, 0, 0, 0, 7]) informs about the primitive field name length. Here, the name of the primitive field is “country” and its size is 7. The next part of binary ([0, 0, 0, 8, 0, 0, 0, 8]) is field value length, which is “Portugal”, and its size is 8. Since the size of field name and value are known, they can be converted to binary. Lastly, [99, 111, 117, 110, 116, 114, 121] and [80, 111, 114, 116, 117, 103, 97, 108] tell us the name of the field and the value of field.

Creating binary is implemented with following rules:

- An integer is always 64 bits, with each byte serialized in sequence. The receiver will recover the integer in the same way.
- Boolean can use only two different tags. There is no size or content, since the tag has all the information needed.
- Strings use a UTF-8 encoding, since it is already byte oriented.
- Structures. Sequence of fields, in which, for each field, you should include: name (a string) and the component proper (serialized according to its type of resource)

Objects with fields should be serialized to a data structure (composed of inner data). This means having a structured data type, with its own tag, and inner components serialized according to their own data type (composition can be recursive).
Chapter 5

Implementation

5.1 Overview

This chapter describes the implementation details of the Binary Level Serialization, Compliance, Conformance, Message Transportation and Deployment. This chapter also gives experimental examples about the new implementation, to show that the solution works in practice and that is better than XML or JSON, regarding data binding and decoupling.

5.2 Binary Level Serialization

In this section, the implementation of the binary level serialization will be presented. The advantages of the binary format, compared to text-based solutions in previous chapters (in Section 4.3), will now be shown with some experimentation.

Computer programming languages have their own object types and special serialization algorithms for their object types, so when you are working with the same language you can easily serialize an object and deserialize it back.

When starting to work with different computer languages and their object types, we cannot use standard Java or .Net(C#) object serialization, because different programming languages use different algorithms for serialization, therefore raising the problem of language-dependent serialization. For example, simple Person object (in Listing 5.1) has a binary representation that is not the same in .Net (Table 5.1) or Java (Table 5.2).

Listing 5.1: Person Object

```java
public class Person {
    public String name;
    public int age;
}
```
Both Java and .Net have different algorithms to serialize and deserialize their object classes, and they are incompatible with each other. The solution for the serialization problem is to avoid using standard serialization libraries for Java and .Net.

### Table 5.1: .Net Serialization Person Object

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>255</th>
<th>255</th>
<th>255</th>
<th>255</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>83</td>
<td>101</td>
<td>114</td>
<td>105</td>
<td>97</td>
<td>108</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>97</td>
<td>116</td>
<td>105</td>
<td>111</td>
<td>110</td>
<td>32</td>
<td>116</td>
<td>101</td>
<td>115</td>
<td>116</td>
<td>44</td>
<td>32</td>
<td>86</td>
</tr>
<tr>
<td>114</td>
<td>115</td>
<td>105</td>
<td>111</td>
<td>110</td>
<td>61</td>
<td>49</td>
<td>46</td>
<td>48</td>
<td>48</td>
<td>46</td>
<td>48</td>
<td>44</td>
<td>32</td>
</tr>
<tr>
<td>67</td>
<td>117</td>
<td>108</td>
<td>116</td>
<td>117</td>
<td>114</td>
<td>101</td>
<td>61</td>
<td>110</td>
<td>117</td>
<td>116</td>
<td>114</td>
<td>97</td>
<td>108</td>
</tr>
<tr>
<td>44</td>
<td>32</td>
<td>80</td>
<td>117</td>
<td>98</td>
<td>108</td>
<td>105</td>
<td>99</td>
<td>75</td>
<td>101</td>
<td>121</td>
<td>84</td>
<td>111</td>
<td>107</td>
</tr>
<tr>
<td>110</td>
<td>61</td>
<td>110</td>
<td>117</td>
<td>108</td>
<td>108</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>83</td>
<td>101</td>
</tr>
<tr>
<td>105</td>
<td>97</td>
<td>108</td>
<td>105</td>
<td>122</td>
<td>97</td>
<td>116</td>
<td>105</td>
<td>111</td>
<td>110</td>
<td>95</td>
<td>116</td>
<td>101</td>
<td>115</td>
</tr>
<tr>
<td>46</td>
<td>80</td>
<td>101</td>
<td>114</td>
<td>115</td>
<td>111</td>
<td>110</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>60</td>
<td>110</td>
</tr>
<tr>
<td>109</td>
<td>101</td>
<td>62</td>
<td>107</td>
<td>95</td>
<td>95</td>
<td>66</td>
<td>97</td>
<td>99</td>
<td>107</td>
<td>105</td>
<td>110</td>
<td>103</td>
<td>70</td>
</tr>
<tr>
<td>101</td>
<td>108</td>
<td>100</td>
<td>20</td>
<td>60</td>
<td>97</td>
<td>103</td>
<td>101</td>
<td>62</td>
<td>107</td>
<td>95</td>
<td>95</td>
<td>66</td>
<td>97</td>
</tr>
<tr>
<td>107</td>
<td>105</td>
<td>110</td>
<td>103</td>
<td>70</td>
<td>105</td>
<td>101</td>
<td>108</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>74</td>
<td>111</td>
<td>104</td>
<td>110</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 5.2: Java Serialization Person Object

<table>
<thead>
<tr>
<th>-84</th>
<th>-19</th>
<th>0</th>
<th>5</th>
<th>115</th>
<th>114</th>
<th>0</th>
<th>23</th>
<th>106</th>
<th>97</th>
<th>118</th>
<th>97</th>
<th>97</th>
<th>112</th>
<th>112</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>105</td>
<td>99</td>
<td>97</td>
<td>116</td>
<td>105</td>
<td>111</td>
<td>110</td>
<td>55</td>
<td>46</td>
<td>80</td>
<td>101</td>
<td>114</td>
<td>115</td>
<td>111</td>
</tr>
<tr>
<td>110</td>
<td>79</td>
<td>-70</td>
<td>94</td>
<td>85</td>
<td>-31</td>
<td>-32</td>
<td>-110</td>
<td>90</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>73</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>97</td>
<td>103</td>
<td>101</td>
<td>76</td>
<td>0</td>
<td>4</td>
<td>110</td>
<td>97</td>
<td>109</td>
<td>101</td>
<td>116</td>
<td>0</td>
<td>18</td>
<td>76</td>
<td>106</td>
</tr>
<tr>
<td>97</td>
<td>118</td>
<td>97</td>
<td>47</td>
<td>108</td>
<td>97</td>
<td>110</td>
<td>103</td>
<td>47</td>
<td>83</td>
<td>116</td>
<td>114</td>
<td>105</td>
<td>110</td>
<td>103</td>
</tr>
<tr>
<td>59</td>
<td>120</td>
<td>112</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>116</td>
<td>0</td>
<td>4</td>
<td>74</td>
<td>111</td>
<td>104</td>
<td>110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After this experiment, creating a common algorithm for different languages was used, instead of a standard serializer for Java or .Net. Then both languages can interoperate by serializing to and deserializing from a common format.

The binary format is always the result of serializing data in each language. Each data has a tag, which describes the data type, the number of bytes that follow and the byte-serialized content. Recovering the serialized data is simply done by testing the tag to find the data type and then using the number of bytes and the serialized content.

A message to be sent from a consumer is an object class (in Java or .Net) that provides a serialization method, which basically builds a serialized message (an array of bytes) by successively adding each of its components, serialized. This is done by invoking the methods of the serialization class for primitive data or by recursively invoking the serialization method of the structured objects that constitute the message.

Instead of every class having their own serializer class, we have implemented the idea of centralizing the “object of primitive type to a sequence of bytes” (and vice-versa) methods into a single class, to avoid the need for every class to have these methods. Since they are static (they receive an object and return bytes, or vice versa), they can simply be concentrated in a static class (no instances) and invoked from anywhere.

The methods to serialize receive a primitive object (integer, Boolean, etc.) and an array of bytes,
returning the array of bytes with the serialized object’s bytes appended. Therefore, each serialization
method grows the byte array. When the user wants to send a message, that message is an instance
of a class that has a method that knows how to serialize it, by invoking the serialization methods of the
static class for each of its variables.

To examine the performance in serializing structured data in binary and text-based data (XML, JSON),
an experiment was designed using the following hardware and software:

- Hardware: IMac (by Apple Inc.) with Intel Core i7 1.7 GHz and 8GB memory.
- Operation System: Mac OS X version 10.11.4.
- Java: version 1.8.

Current version of object serialization libraries were selected shown in the following:

- JAXB Serializer for XML serialization.
- Jackson Serializer for JSON serialization.
- OpenEXI for XML compression

The experiment was designed as follows:

- Ten kinds of query were prepared for the weathercast provider. They were queries with ten different
  size of weather: 0, 100, 200, 300, 400, 500, 600, 700, 800, 900 weathercast query.
- The serialized file was measured and the execution time was measured using System.currentTimeMillis()
  shown in Listing 5.2.

```
Listing 5.2: Serialization program for testing
1   long start = System.currentTimeMillis();
2   Query q = new Query();
3   for (int i = 0; i < 900; i++) {
4       q.weathers.add(new Weather());
5   }
6   json = serialize(query);
7   long end = System.currentTimeMillis();
8   double time = (double) (end-start);
```

The average size of ten kinds of serialized files is given in Table 5.3.

From the point of view of the average size, the largest serialized file is obtained using XML, but EXI
(Efficient XML Interchange), which uses compression technology, has the best serialized file size when
compared with others. But it does not reduce the parsing time, since text needs to be recovered. On
the other hand, Binary and JSON have very similar serialized file size. From the perspective of execution
time, binary serialization spends less time when compared with the others as seen in Figure 5.1.
Regarding quantitative aspects, the size of binary-based serialized data is better than XML-based and
JSON-based serialization, since there is no schema required and also in terms of data binding, the
binary-based serialization gives us a big advantage by removing stub generation and DOM inspection.
Table 5.3: Average sizes (in bytes) of several resource representations.

<table>
<thead>
<tr>
<th>Format</th>
<th>Average size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>62008</td>
</tr>
<tr>
<td>EXI</td>
<td>3030</td>
</tr>
<tr>
<td>JSON</td>
<td>32036</td>
</tr>
<tr>
<td>Binary</td>
<td>35984</td>
</tr>
</tbody>
</table>

**Figure 5.1: Average Execution times for serialization of XML, JSON and Binary**

![Graph showing execution times for serialization of XML, JSON, and Binary formats.](image)

### 5.3 Receiver and Compliance

Compliance is done at the binary level, with primitive components. It is checked between the received message and the formal argument of the receiver’s method. Only the components that match are assigned to the formal argument of the operation. Two partners will be able to communicate as long as the sender complies with the receiver and the receiver conforms to what the sender expects, and it supports all the features that the sender requires.

When a suitable operation is found, the server will complete operation and create a response for the client. The system also supports optional components. Therefore, the serialization methods in the static serialization class should include the name of the component, whether it is mandatory or not (with annotation), the type (encoded in the tag) and the value. Each primitive data type can have a mandatory annotation. Messages sent use only the mandatory values. Serialized formal arguments can be either mandatory and non-mandatory. The receiver has always a default value for the message, so for the data that don’t have mandatory annotations, it will always use the default values.

Let us explain that with an experiment, consisting of two consumers (one developed with .Net (C#) and another with Java) and two providers (one developed with .Net (C#) and another with Java). The general properties of the two consumers can be seen in Table 5.4, and the properties of two providers can be seen in Table 5.5.

A Windows Mobile application (in Listing 5.3) sends two different query messages (WeatherQuery1
as seen in Listing 5.7 and WeatherQuery2 as seen in Listing 5.8) to Cloud1 and Cloud2 providers. Android Mobile application (in Listing 5.4) sends a query message(WeatherQuery3, as seen in Listing 5.9) to Cloud 1 and Cloud 2 providers. If the receiver conforms to what the sender expects, then there is compliance, the received message is partially assigned to that argument, and the operation is invoked.

**Listing 5.3: Windows Mobile application**

```java
class Program
{
    static void Main(string[] args)
    {
        Query1 q = new Query1();
        Message msg = new Message(q);
        string msgToSend = msg.Serialize();
        JavaWebServiceClient service = new JavaWebServiceClient();
        // Show result from server
        Console.WriteLine("Result From Server: \n" +
            service.GetResult(msgToSend));
        Console.ReadLine();
    }
}
```

**Listing 5.4: Android Mobile application**

```java
public static void main(String[] args) throws Exception {
    final ChatClientEndpoint clientEndPoint = new ChatClientEndpoint(new URI("ws://javatomcatthesis.azurewebsites.net/JavaWebServerWebSocket-1.0/javawsendpoint"));
    clientEndPoint.addMessageHandler(new ChatClientEndpoint.MessageHandler()
    {
        public void handleMessage(String message) {
```
System.out.println(message);

Query3 q = new Query3();
Message msg = new Message(q);
byte[] msgToSend = msg.SerializeBinary();
clientEndPoint.sendMessage(msgToSend);
Thread.sleep(30000);

Listing 5.5: .Net provider AvailableMethod notation in Receiver Class

namespace CSharpWebServer.istesuysal.thesis
{
    public class Receiver
    {
        [AvailableMethod]
        public void AvailableMethod(Weather1 weather1)
        {
        }
        [AvailableMethod]
        public void AvailableMethod(Weather2 weather2)
        {
        }
        [AvailableMethod]
        public void AvailableMethod(Weather3 weather3)
        {
        }
    }
}

Listing 5.6: Java provider @AvailableMethod notation in Receiver Class

package ist.ionesuysal.thesis;

public class Receiver
{
    @AvailableMethod
    public void AvailableMethod(Weather4 weather4)
    {
    }
    @AvailableMethod
    public void AvailableMethod(Weather5 weather5)
    {
    }
    @AvailableMethod
    public void AvailableMethod(Weather6 weather6)
    {
    }
}
Providers have operations with at most one parameter, which can be structured. Matching will be done between the message from consumer and the input parameter of operations. Input parameters of operations in each server can be seen in Table 5.6.

Table 5.6: Information about input parameter of operations in the provider for experiment.

<table>
<thead>
<tr>
<th>Cloud Server 1</th>
<th>Cloud Server 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weather1 (in Listing 5.10)</td>
<td>Weather4 (in Listing 5.13)</td>
</tr>
<tr>
<td>Weather2 (in Listing 5.11)</td>
<td>Weather5 (in Listing 5.14)</td>
</tr>
<tr>
<td>Weather3 (in Listing 5.12)</td>
<td>Weather6 (in Listing 5.15)</td>
</tr>
</tbody>
</table>

---

Listing 5.7: Details of WeatherQuery1 Object

```java
public class WeatherQuery1
{
    public String country = "Portugal";
    public String city = "Lisbon";
}
```

Listing 5.8: Details of WeatherQuery2 Object

```java
public class WeatherQuery2
{
    public int cityCode = 35121;
}
```

Listing 5.9: Details of WeatherQuery3 Object

```java
public class WeatherQuery3
{
    public double latitude = 38.736946;
    public double longitude = -9.142685;
}
```

---

Listing 5.10: Details of Weather1 Object

```java
public class Weather1
{
    [Mandatory]  
    public String country = "Portugal";
    [Mandatory]  
    public String city = "Braga";
}
```

Listing 5.11: Details of Weather2 Object

```java
public class Weather2
{
```

43
public int cityCode = 35121;
public String country = "Portugal";
public String city = "Lisbon";
}

Listing 5.12: Details of Weather3 Object

public class Weather3
{
  [Mandatory]
  public String country = "Portugal";
  [Mandatory]
  public String city = "Braga";
  public int cityCode = 351253;
  public double latitude = 41.530918;
  public double longitude  = -8.780565;
}

Listing 5.13: Details of Weather4 Object

public class Weather4
{
  @Mandatory
  public String country = "Portugal";
  @Mandatory
  public String city = "Lisbon";
  public double latitude = 38.736946;
  public double longitude  = -9.142685;
}

Listing 5.14: Details of Weather5 Object

public class Weather5
{
  @Mandatory
  public double latitude = 38.736946;
  @Mandatory
  public double longitude  = -9.142685;
}

Listing 5.15: Details of Weather6 Object

public class Weather6
{
Let us start with Windows mobile application (Mobile 1) to explain the compliance and conformance concepts, by providing a message with a WeatherQuery1 object (in Listing 5.7) and sending the WeatherQuery1 object to the .Net cloud provider (Cloud Server 1). The Weather1 object in Cloud 1 provider will not match WeatherQuery1, because the Weather1 object in Cloud 1 provider has two mandatory fields, as seen in Listing 5.10. The Country and city fields in the Weather1 object do not have same value as the country and city fields in the WeatherQuery1 object. The Weather2 object in Cloud 1 provider has three fields (one mandatory and two optional), as seen in Listing 5.11. Since the mandatory field is not in the WeatherQuery1 object, matching will not occur. The Weather3 object in Cloud 1 provider will also not match the WeatherQuery1, because the Weather3 object in Cloud 1 has two mandatory fields and three optional fields, as seen in Listing 5.12, but the mandatory city field does not have the same value as the city field in the WeatherQuery1 object. Since no suitable operation is found, the server will not invoke an operation and will not create a response for consumer in Cloud Server 1.

Next, let us try sending a WeatherQuery1 object to Cloud Server 2 to check if there is any compliant operation. The Weather4 object in Cloud 2 provider will match the WeatherQuery1 object, because Weather4 object in Cloud 2 has two mandatory fields and two optional fields, as seen in Listing 5.13. The City and country fields in the Weather4 object have the same value as city and country fields in the WeatherQuery1 object. The optional fields in the Weather4 object do not exist in the WeatherQuery1 object, but that is not important since they are optional, and they will be used with their default values. Now a suitable operation is found and the server will invoke it, sending a response message to the consumer.

Now, we will send the WeatherQuery2 object (in Listing 5.8) with Windows mobile application to the Cloud 1 and Cloud 2 providers. The Weather1 object in Cloud 1 provider will not match the WeatherQuery2 object, because the Weather1 object has two mandatory fields and these fields do not exist in the WeatherQuery2 object. The Weather2 object in Cloud 1 provider will match the WeatherQuery2 object because the Weather2 object in Cloud 1 provider has one mandatory field and two optional fields. The cityCode field in Weather2 object has the same value as the cityCode field in the WeatherQuery2 object. The optional fields in Weather2 object do not exist in WeatherQuery2 object, but that is not important since they are optional, and they will be used with their default values. The WeatherQuery2 message complies with the Weather2 object. The server will invoke the operation and send a response to the consumer.

As explained above, sending a WeatherQuery1 message in Windows phone application will comply with the Java weathercast provider and will return a response message. In the same way, the WeatherQuery2 message in Windows phone application will comply with the .Net weather provider and will return result a response message.

The consumer does not need to be written in .Net. It can be written in Java or any other language.
We can also show the implementation of the Android phone application (Mobile 2). This application has the WeatherQuery3 object, as seen in Listing 5.9, and it will send that query message to Cloud 1 and Cloud 2 providers. Because of reasons explained before, any operation in Cloud 1 provider will not match the WeatherQuery3 object because mandatory fields of the Weather1, Weather2 and Weather3 objects do not exist in the WeatherQuery3 object. Any operation in Cloud 2 provider will also not match the WeatherQuery3 object, because again mandatory fields of the Weather4, Weather5 and Weather6 objects do not exist in the WeatherQuery3 object.

The results that are explained above about partial assignment of Mobile 1 and Mobile 2 applications can be seen in Table 5.7 and in Table 5.8.

<table>
<thead>
<tr>
<th>Weather1</th>
<th>Weather2</th>
<th>Weather3</th>
<th>Weather4</th>
<th>Weather5</th>
<th>Weather6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Query 2</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Query 3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.8: Compliance and Conformance between the Cloud Servers and Mobile phones.

<table>
<thead>
<tr>
<th>Mobile 1</th>
<th>Query 1</th>
<th>Cloud Server 1</th>
<th>Cloud Server 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Query 2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Mobile 2</td>
<td>Query 3</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The demonstration in this section intends to show that the proposed solution is an alternative to XML, JSON, WSDL and REST. The example presented here involves compliance or conformance. It allows changes to the client and to the server without having to inform or change both sides.

### 5.4 Message Transportation

Transferring the array of bytes from the sender to the receiver requires a binary channel such as Web Sockets or HTTP/2, or a more classical way by encoding and decoding the binary array with BASE64 and then using typical HTTP-based solutions (Web Services or REST). The classical solution is non-optimal compared to the other ones, but it is easier to implement with existent tools.

The message transportation in this new solution is done essentially with JavaScript and Web Sockets (Figure 5.2), to circumvent some of the limitations of HTTP/1.1. Nevertheless, the solution also supports the classical way by encoding and decoding the binary array with BASE64 and then using typical HTTP-based solutions.

Web Sockets constitute the technology chosen for the implementation, because they are fundamental in the efficient support for binary data. Web Sockets remove binary data restrictions and increase performance. They use the protocol upgrade feature of HTTP and provide a substantial degree of compatibility with existing systems. Now they are part of the HTML5 world, servers and firewalls. They are increasingly supported.
To examine the performance in HTTP-based solutions and Web Sockets, an experiment was designed using the following hardware and software:

- **Hardware**: IMac (by Apple Inc.) with Intel Core i7 1.7 GHz and 8GB memory.

- **Operation System**: Mac OS X version 10.11.4.

- **Java**: version 1.8.

The experiment was designed as follows:

- Ten kinds of query were prepared for weathercast provider. They were queries with 6 different size of weather message: 10, 100, 500, 1000, 5000, 10000 weathercast query message.

### Table 5.9: Average times (in ms) of HTTP-based solutions and Web Socket.

<table>
<thead>
<tr>
<th>Nb of Message</th>
<th>Web Service (HTTP) (in ms)</th>
<th>WebSocket (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
<td>19</td>
</tr>
<tr>
<td>500</td>
<td>520</td>
<td>69</td>
</tr>
<tr>
<td>1000</td>
<td>1050</td>
<td>116</td>
</tr>
<tr>
<td>5000</td>
<td>5180</td>
<td>520</td>
</tr>
<tr>
<td>10000</td>
<td>10520</td>
<td>1019</td>
</tr>
</tbody>
</table>

As seen in Table 5.9 and in Figure 5.3, there is a big difference between the HTTP-based solutions and Web Sockets. It is clear to see that using Web Sockets technology increases performance. Web Sockets also remove the limitations of HTTP/1.1, adding binary support.

5.5 Deployment

The new solution is developed in two different programming languages, .Net and Java. The solution is deployed to Microsoft Azure Cloud, chosen instead of other providers because Microsoft provides free access to their application servers of Microsoft Azure Cloud with a student subscription account. Another reason, .Net and Java technologies can be deployed using the same platform. Microsoft Azure support Java and .Net, so two different providers will be deployed on the same platform. The Azure application servers support Web Sockets, which is also a benefit when testing the new solution using Web Sockets in a cloud environment. The .Net client can send a message to a Java provider over the cloud by using Web Sockets technology, and similarly a Java client can send a message to a .Net provider. Using Microsoft Azure Cloud technologies allowed us to test the project in the cloud environment. The characteristics of the Java application server and of the .Net application server can be seen in Figure 5.4 and in Figure 5.5, respectively.
Figure 5.5: The characteristics of .Net application server.
Chapter 6

Comparison with existing technologies

6.1 Overview

This chapter compares the current technologies and the new implemented approach. This comparison is done qualitatively and quantitatively. The aim of this chapter is to summarize the advantages and disadvantages of the new approach, compared to the current technologies.

6.2 Qualitatively and Quantitatively Comparison

6.2.1 Textual or binary representation

Current integration technologies for distributed systems in cloud environments are generally based on textual data description languages (e.g., XML and JSON) and the HTTP protocol. These technologies were especially designed for human-level interaction, but create integration problems at the application level.

As described above, SOA is usually implemented by Web Services with WSDL. WSDL is a set of conventions on XML usage to describe services at the interface level and SOAP as a message protocol, which is again based on XML. Many developers find SOAP cumbersome and hard to use. For example, working with SOAP in JavaScript means writing tons of code to perform extremely simple tasks because you must create the required XML structure absolutely every time. One perceived disadvantage is the use of XML because of the verboseness of it and the time it takes to parse.

REST requires that data types, which are usually called media types, be standardized or previously agreed upon, when they are application specific. REST doesn’t have to use XML to provide the response. You can find REST-based Web services that output data in Comma-separated values (CSV)[26], JavaScript Object Notation (JSON) and Really Simple Syndication (RSS). The point is that you can obtain the output in a form that you need. That is why it is easy to parse with the language you need for
your application. While this may seem to add complexity to REST due to the need of handling multiple
formats, JSON usually is a better fit for data and parses much faster. REST allows better support for
browser clients due to their support for JSON.

SOA and REST use textual representation. Text has been touted as human readable and therefore
it seems advantageous over binary, but this is true only for very simple documents, especially when
using XML. By the way, textual representation brings parsing overheads and poor support for binary
data. When using SOA and REST solutions you need to produce a client stub, Schema validation and
also DOM parsing. All these overheads are a big deal for performance and create complexity in terms
of usability. Instead of using textual representation, the binary representation provides native support for
binary data, has a smaller length and is faster to parse. As seen in Section 5.2 with Figure 5.1, using
binary data has more advantages than using XML or JSON.

The implemented approach in the new solution is designed to work with binary representation. The
binary representation uses a modified version of the TLV format (Tag, Length and Value) used by ASN.1
[25]. This not only supports the direct integration of binary information but also facilitates parsing, since
each resource is primitive or structured and follows a simple tag scheme.

6.2.2 Message protocol

The approach implemented in the thesis does not depend on any particular transport protocol, relying
only on message delivery. Any existing server can be used, based on HTTP, Web Sockets or any other
protocol. In fact, several servers can be used simultaneously, receiving messages that are handed over
to the message handlers that are able to process them. HTTP/1.1 still can be used, but BASE64 must
be used to overcome the limitation of HTTP/1.1 of not supporting binary data. Everything is serialized
to the binary, then encoded in BASE64 and decoded at the receiver to recover the binary data. Using
new protocols, such as Web Sockets or HTTP/2, reduces some of these problems. Web Sockets, now
part of the HTML5 world, remove this restriction, add binary support and increase performance. Using a
platform which uses WebSocket or HTTP/2 will increase usability and performance regarding message
transportation.

As seen in section 5.4, there is a big difference in terms of performance using Web Sockets in-
stead of using HTTP-based solutions. The implemented solution in this thesis has the big advantage of
supporting and being able to exploit new protocols such as Web Sockets or HTTP/2.

6.2.3 The interoperability problem

As seen in previous chapters, in both SOA and REST interoperability is achieved by using common
data types (usually structured), either by sharing a schema (i.e., WSDL files) or by using a previously
agreed upon data type (typical in RESTful applications). This is provided with full interoperability, and
there is usually no support for partial interoperability and polymorphism in distributed systems with SOA
and REST. Asymmetric interoperability allows to show that interaction is still possible with only a partial
knowledge of types, as long as the characteristics actually used, which are included by supporting
partial interoperability. The trick is to allow partial interoperability, by considering only the intersection between what the consumer needs and what the provider offers. It allows for increased interoperability, adaptability and changeability, without the need to have resource types necessarily shared or previously agreed upon. The following sections, (section 6.2.4 and 6.2.5), will provide information and comparison between full interoperability and partial interoperability.

### 6.2.4 Full interoperability in SOA and REST

The basis of data interoperability with XML and JSON is schema sharing (at runtime or with previously standardized or agreed-upon Internet media types). Both the producer and consumer (reader) of a document should use the same schema, to ensure that any document produced (with that schema) can be read by the consumer. This means that both producer and consumer will be coupled by this schema. These solutions use full interoperability, in which the consumer can use all the characteristics that the provider exposes. Schemas must be shared between interacting Web Services, establishing coupling for all the possible values satisfying each schema, even if they are not actually used. Additionally, you cannot change the schema without informing both client and server. So you cannot change either the client or the server without breaking interoperability. For example, in case of SOA style, if the producer changes some operation without informing the consumer, then the producer will have different WSDL schema than consumer currently has, because WSDL schema shows which operations are available and how data should be structured to send to those operations. Since the WSDL schema is different between producer and consumer, the consumer will not communicate with the producer. Another example, thinking that there are a producer and two consumers that call operations of this producer, but they do not use same operations, because they do different tasks, even if using the same producer. Still, they need to have the same WSDL schema even they don’t use all the characteristics of provider and in case if the producer changes some operations for one consumer, then both consumers need to be informed and changed, since they use the same schema.

Another issue, when you expose a method as a Web service, the data contained in the programming language object must be converted to a language-independent format - namely XML. In the other direction, the XML must be converted into the expected object in the producer (as seen in Figure 6.1). Data binding is all about converting from XML to programming-language specific data structures. Web service tools help by generating code that performs this mapping. The code generation process takes a WSDL as input and generates client code in a particular programming language, referred to as a “stub”, which can access the Web service described by that WSDL. Also, it can generate server-side code, referred to as a “service skeleton”. Data binding frameworks use SAX or DOM to read and write XML documents. Solving data interoperability with data binding, stub and skeleton creation is inefficient with a waste of time, processor power and memory.

Basically, in these solutions, the receiver needs to be worried with the format and field names of the incoming messages that consumer creates. The target of asymmetric interoperability is removing this coupling problem between consumer and producer.

53
6.2.5 Asymmetric interoperability

The implemented approach of the new solution proposes to use partial interoperability, based on the concepts of compliance and conformance. It introduces a different perspective, stronger than similarity but weaker than commonality (schema sharing). The trick is to allow partial interoperability, by considering only the intersection between what the consumer needs and what the provider offers. It allows for increased interoperability, adaptability and changeability, without the need to have resource types necessarily shared or previously agreed upon. Building interoperability on compliance and conformance avoids the problem of having to define schemas as separate documents and to agree upon them beforehand. As long as compliance and conformance hold, any two resources can interoperate, even if they were designed unaware of each other. In that solution, a consumer will communicate with a provider as in current classical solutions, but with a different perspective that avoids schema sharing. Since there is no shared schema, both consumer and producer can change their structure without informing each other. The consumer and producer do not need to have the same structure, and matching will be done field by field between the message and the service operation’s formal argument. With asymmetric interoperability the receiver deals with its own format and field names, and there is no need to generate a stub to deal with the message (Figure 6.2).

The compliance-based assignment of the message to the receiver’s formal argument is made at binary level, field by field. The receiver deals only with the message format and field names it already knows, instead of having to deal with the whole structure of the message and its field names. This is how coupling is reduced, as long as compliance holds. This is the main advantage of asymmetric interoperability. If a consumer tries to communicate with a producer and there is no match between the

![Figure 6.1: Data interoperability.](image-url)
message and the service operation’s formal argument then no operation will be invoked, which means that the consumer and producer will not exchange the information.

As seen in the example of section 4.2, the client or the server can change their structure of messages without breaking interoperability. This means that both client and the server reduce coupling, when compared with current solutions. The example (in Section 4.2) also shows that using the asymmetric compliance and structured data is equally suitable for both examples of SOA and REST (in Section 3.3), allowing both service interfaces and structured resources.

Figure 6.2: Asymmetric interoperability.
Chapter 7

Conclusions

7.1 Achievements

In this thesis, the proposed solution for a new programming technology is an alternative to current solutions, which solve the interoperability problems by sharing data schemas. This entails a coupling problem between the provider and the consumer, because both are forced to implement full interoperability. The proposed solution provides the maximum decoupling possible, while ensuring the minimum interoperability requirements, by using compliance and conformance instead of sharing the same schema. As long as compliance and conformance hold, any two resources can interoperate, even if they were changed unbeknownst to each other. This solution allows changing the structure of the client and the server.

Current solutions use XML or JSON as a data type for sending or receiving a message. They are based on text, heavy, hard to parse and costly in communications. There are technologies for the binary solutions of XML or JSON format, but they still need data to be compressed and decompressed. Using XML and JSON forces both sides to check and validate their schemas to guarantee that each arrived message is in a correct format. Validation of XML is inefficient in computer terms due to parsing data and to build a DOM tree where all characters of a component need to be parsed to reach the next one, leading to high memory consumption. The proposed solution uses the binary data format natively. It does not require a separate schema and also increases performance. The binary message format results from compilation of the source program and uses self-description information only when needed. This allows maintaining all the necessary information to communicate in a standard and platform-independent way.

Typical current solutions use a connectionless protocol (HTTP/1.1), which lacks native support for binary data. The solution has been designed to use new protocols such as Web Sockets and HTTP/2, which support binary data and increase performance. However, the classical HTTP protocol can still be used in the solution, by resorting to BASE64 encoding.
7.2 Future Work

Although the provided analyses and methodologies are quite good and show that new solution can be a replacement for existing ones, there are some possible future developments that could be made to improve the implemented work, namely:

- The solution currently has some limitations such as structures as compound resources without operations (only data fields). So, requests and responses must be either primitive resources (e.g., integers) or structures (not complete resources). It would be desirable to add the functionality of operations instead of just data fields.

- The interoperability framework presented in this work needs to be improved and should be completed.

- The compliance and conformance algorithms are implemented in the solution, but they need to be optimized to increase the performance.

- There is a need to create a plan and implement some quality tests, to ensure that all its functionalities are implemented correctly, and possible development bugs are dealt with.
Bibliography


