Distributed Programming in Cloud Computing Platforms

Enes UYSAL

Master Project in Information Systems and Computer Engineering
Alameda
Instituto Superior Tecnico

(January 2015)

Abstract

The present document is the follow up solutions of distributed programming for interoperability between heterogeneous systems essentially are SOA or REST technology and a new distributed service programming language, SIL (Service Implementation Language) which already existing and developed for programming distributed cloud computing environment. To successfully accomplish this goal it is important to understand more traditional solutions based on SOA (Web Services) and REST technologies then combine them to maximize advantages and to minimize the disadvantages. This document will focus on a new language which already existing and developed for programming distributed cloud computing environment, comparing this solution with more traditional solutions based on SOA (Web Services) and REST technologies. Having that, it is crucial to understand a language to use in this work which uses binary data format natively, does not require schemas and the runtime uses Websockets, much more efficient than HTTP and also has native support for dynamic platforms such as clouds and sensor networks. In this way, we will see the differences of SIL architecture from others. A brief overview over the SIL architecture will demonstrate how it is different from current solutions and adaptability for cloud platforms. Finally, ways to evaluate both SIL architecture and the traditional solutions based on SOA (Web Services) and REST technologies as a whole will be discussed.

Keywords

Cloud computing, SIL, Service Implementation Language, SOA, REST, XML, JSON, Distributed programming

Introduction

This thesis proposes a comparison between current solutions and a new solution of programming of distributed applications, with all the basic interoperability problems involving distributed platforms and heterogeneous components. This document is structured logically to allow the reader to understand the problem addressed and its challenges, followed by a clarification of the concepts involved and their pertinence in the market today, and ending in the solution proposal and evaluation methodology as a result of the work done.

The current chapter contextualizes the problem description, thesis's theme and identifies the research challenges and objectives in this work.

Problem Description

Distributed applications are a necessity in most central application sectors of the contemporary information society, including e-commerce, e-banking, e-learning, e-health, telecommunication and transportation. This results from a tremendous growth

of the role that the Internet plays in business, administration and our everyday activities. The fundamental problem is the programming of distributed applications with all the basic interoperability problems involving distributed platforms and heterogeneous components.

On the other way, Clouds constitute a new challenge for distribution. They bring new opportunities such as scalability, dynamic instantiation, location independence (including migration), application management and multi-tenancy (several users using the same application independently). However, they constitute distributed platforms and that's why they need to be able to support distributed applications as easily as possible.

The application services hosted under Cloud computing model have complex provisioning, composition, configuration, and deployment requirements. Evaluating the performance of Cloud provisioning policies, application workload models, and resources performance models in a repeatable manner under varying system and user configurations and requirements is difficult to achieve.

In the next sections we will describe the most popular current solutions and our new solution to overcome this problem.

Current solutions

The main integration technologies available today are based on Web technologies, namely HTTP, XML, JSON, Web Services using either SOA or REST architectural styles, with Web Services and REST on HTTP. They do not always meet expectations for distributed cloud computing environment because they generate two different approaches to the application integration problem for example Web Services are a closer modelling match for more complex applications, since they allow a service to expose an arbitrary interface. In turn they lack structure (all services are at the same level) and are complex to use. These technologies were conceived for human browsing or large-scale application components, complex and heavy. REST is lighter and with a smaller component grain, but entails a significant modelling difference between real world components and programming components, meaning that its simplicity is not easy to extend to extend to more complex, enterprise-class applications. In addition, both SOA and REST rely, in practice and in most cases, on HTTP and text based data (XML or JSON). These solutions will be explained more in depth in section 2 and 3.

New Solution

This thesis focuses on developing research work on the subject to understand traditional solutions based on SOA (Web Services) and REST technologies then combine them to maximize advantages and to minimize the disadvantages.

We want to investigate a new programming technology that constitutes a distributed programming language. Web Services and RESTful APIs are integration interfaces

but SIL is a programming technology that entails meta-model that unifies services and resources. It has native support for dynamic platforms such as clouds and sensor networks. The result is a solution for the entire interoperability stack, namely protocol, syntax, semantics and behaviour. This combines the features of SOA, REST and programming languages and does not require a specific protocol. And also understanding SIL architecture shows how it is different from current solutions. Section 4 will give more information related SIL architecture.

Goals

The motivation of this work is to contribute to improve researching concepts and best practices related SIL architecture and in our dissertation we will discuss following subjects;

- Current solutions and their limitations;
- SIL solution and its advantages;

Then we will present examples to assess and compare qualitatively the existing solutions and SIL to show that SIL is simpler and a better modelling tool than REST but has also it has the capability of implementing complex applications that characterizes SOA. For cloud computing we will implement an example in SIL that assumes an elastic computing and measuring the execution time with a varying number of computing nodes that will help us to show the advantages of SIL in the cloud environment. To resolve accessing a cloud platform with a significant number of nodes, we will use a cloud simulator, NetworkCloudSim, which allows more accurate evaluation of scheduling and resource provisioning policies to optimize the performance of a Cloud infrastructure [13].

The work plan for future work of dissertation can be found in Fig. 6

As a related work In Section 2 the concept of web services is described more in depth and SOA and REST technologies are explained. In section 3 Study of SOA and REST is presented. In section 4, solution architecture and the starting point for the development of the tool as well as some starting measures for further revision is presented. In section 5 we describe the scheduling of future work. Section 6 contains conclusion of this report.

About Web Services

This section explains concept of web services and approaches to integrate applications for interoperability problem.

There have been different approaches to integrate applications for interoperability problem. Nevertheless, four main integration styles [1] (see Figure 1) can be distinguished;

Batched file transfer: A very easy to implement style, since all programming language and systems have mechanisms to manipulate files. The most important is choosing a format to be shared by all applications, usually a standard format such as XML. Despite applications remain decoupled (they can be changed as long as they

respect data format) the most important drawback is the update frequency, which can derive into data inconsistencies between different applications. It can make extra work for developers: choosing where to put files, establishing when to delete old files, which files names are to be used or implementing a locking system to avoid concurrent file updates.

Shared database: Shared database is an extended solution which has some clear benefits. Firstly, most databases use SQL to interact with them, which is a common known language for developers and a well-supported language in any platform. There is no chance for any data inconsistency (even multiple updates are handled by transactions mechanisms). However, although it is positive facing future semantic problems before integrating applications, it also implies delays for application developers who may not like to assume them and can just separate from integration. In addition to this, trying to integrate third party tools can be difficult because of these products can have limited adaptability. Finally, depending on the amount of tools being integrated into a shared database, it can become a performance bottleneck (even with distributed databases).

Remote procedure call (RPC): This style focuses on sharing functionality. Instead of offering non-encapsulated data (such as in shared database style, where all applications know the details of the database schema), applications share interfaces to functionalities that wrap shared data (e.g.: to retrieve or modify data, to launch specific actions, or even supporting different interfaces for the same data to different users). This reduces coupling to a given data structure but also creates coupling with regards to service orchestration (in which order should functionalities be consumed to do complex interactions). An additional drawback to this style is it requires synchronicity, that is, if a service is unavailable, the request will be lost; this introduces some issues on reliability and performance.

Message bus: Asynchronicity is the most relevant characteristic of this style. Applications send messages which can be broadcasted, sent to a specific application or any other scenario. This style is similar to File transfer style, although data transmitted is usually small and sending frequency high.

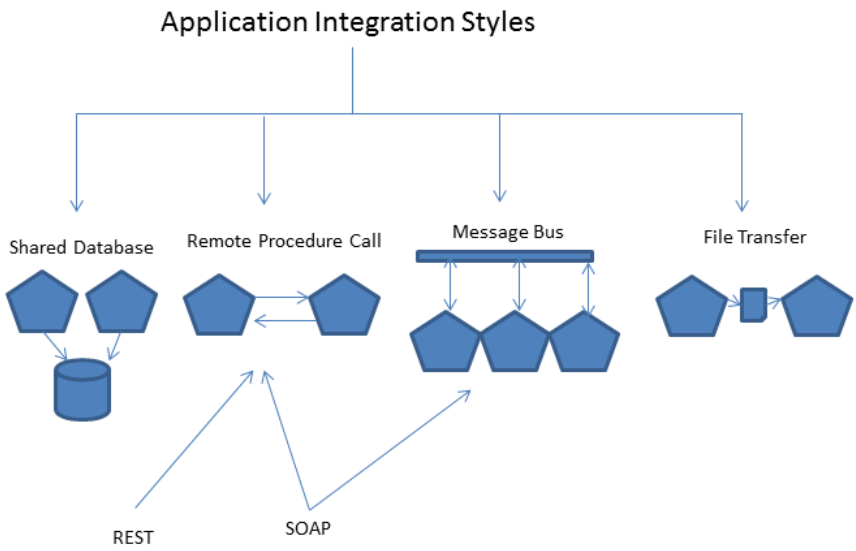

Figure 1 - Application Integration Styles

In this project we will discuss most common integration technologies available t day are based on the SOA and REST architectural styles.

A web service is a piece of business logic, located somewhere on the Internet, that is accessible through standard based Internet protocols such as HTTP or SMTP. Using a web service could be as simple as logging into a site or as complex as facilitating a multi organization business negotiation.

A web service has special behavioural characteristics like text based, loosely coupled, coarse grained, ability to be synchronous or asynchronous, supporting Remote Procedure Calls (RPCs) and supporting document exchange. The general process of engaging a Web Service can be described as Figure 2:

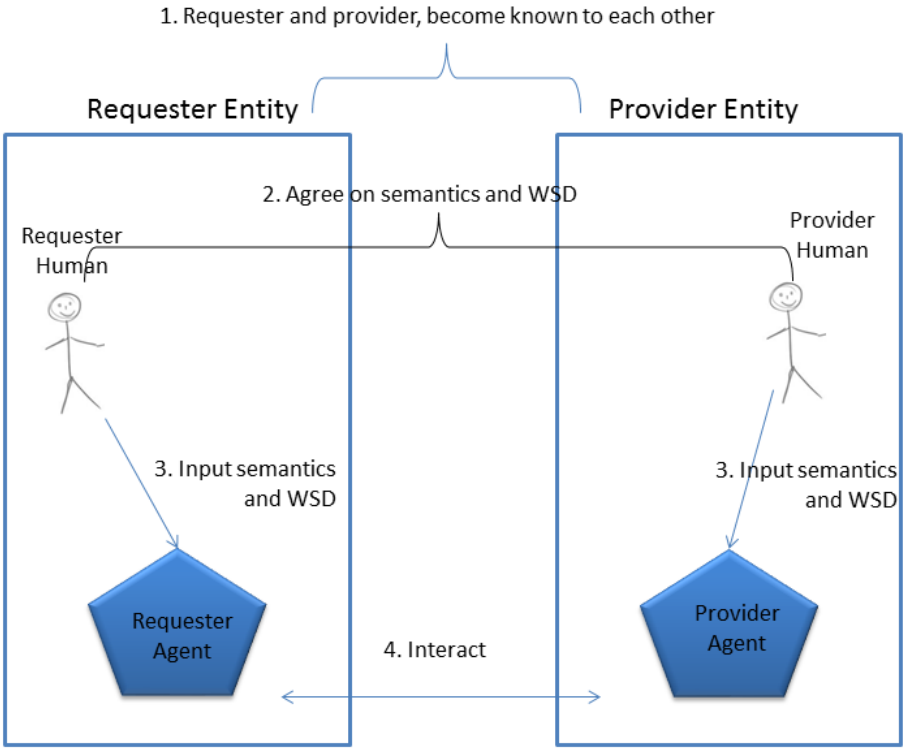

Figure 2 - The General Process of Engaging a Web Service

1. Entities, requester and provider, become known to each other (although it is more usually that only provider entity becomes known by means of a service registry or service broker);
2. There is an agreement (again, service registry supports this agreement, by means of allowing the provider to registry a description of the service, which requester may examine to see if it fits his needs);
3. Agents receive WSD and semantic as input, to realise the service;
4. Interaction starts by means of message exchanging between the requester and provider agents, who represent the requester and provider entities.

A Web service supports direct interactions with other software agents using text based messages exchanged via Internet based protocols [2]. The most common approaches to implement Web services are simple object access protocol (SOAP) and representational state transfer (REST), which can also use text messages, but uses HTTP as the message protocol.

SOAP used the new Extensible Mark-up Language (XML) [16] as a packaging format for a Remote Procedure Call (RPC) mechanism, and thus simply used the

well-established pattern of using RPC mechanisms for implementing distribution, and packaged it using the Web technologies XML and HTTP. While questions around SOAP's ability to implement true "Web Services" (instead of just implementing RPC over the Web) surfaced relatively early [17], SOAP had already gained considerable momentum and most major vendors had joined the standardization process of SOAP and related technologies. Representational State Transfer (REST) [21] as a posthoc conceptualization of the Web as a loosely coupled decentralized hypermedia system was coined as a term in 2000, but it took several years until it became clearly visible in the mainstream that the model of "Web Services are based on SOAP" had a serious competitor in the form of services that better conformed to the architectural principles of the Web. Because those principles were defined by the REST model, this new variety of Web Services often was referred to as "RESTful Web Services."

In next section we will discuss these popular Web Service technologies.

SOAP

The Simple Object Access Protocol, or SOAP, is the latest in a long line of technologies for distributed computing. It differs from other distributed computing technologies in that it is based on XML, and is designed for the exchange of information in a distributed computing environment and also that thus far it has not attempted to redefine the computing world [3].

Instead, the SOAP specification describes important aspects of data content and structure as they relate to familiar programming models like remote procedure calls (RPCs) and message passing systems. There is no concept of a central server in SOAP; All nodes can be considered equals, or even peers.

The protocol is made up of a number of distinct parts. The first is the envelope, used to describe the content of a message and some clues on how to go about processing it. The second part consists of the rules for encoding instances of custom data types. This is one of the most critical parts of SOAP: its extensibility. The last part describes the application of the envelope and the data encoding rules for representing RPC calls and responses, including the use of HTTP as the underlying transport.

RPC style distributed computing is the most common Java usage of SOAP today, but it is not the only one. Any transport can be used to carry SOAP messages, even though HTTP is the only one described in the specification.

The SOAP specification requires a SOAP implementation to support the transporting of SOAP XML payloads using HTTP, so it's no coincidence that the existing SOAP implementations all support HTTP. SOAP can certainly be used for request/response message exchanges like RPC, as well as inherently one-way exchanges like SMTP. HTTP is a natural for an RPC style exchange because it allows the request and response to occur as integral parts of a single transaction.

When sending data over a network, the data must comply with the underlying transmission protocol, and be formatted in such a way that both the sending and receiving parties understand its meaning. This is what we refer to as data encoding. Data encoding encompasses the organization of the data structure, the type of data transferred, and of course the data's value. Just like in Java, it is the data that gets serialized, not the behaviour.

Data encoding and serialization rules help the parties involved in a SOAP transaction to understand the meaning and content of the message. The model for SOAP encoding is based on XML data encoding, but the encoding constraints or alters those rules to fit the intended purpose of SOAP.

All SOAP messages are packaged in an XML document called an envelope, which is a structured container that holds one SOAP message. The metaphor is appropriate because you stuff everything you need to perform an operation into an envelope and send it to a recipient, who opens the envelope and reconstructs the original contents so that it can perform the operation you requested. The contents of the SOAP envelope conform to the SOAP specification, allowing the sender and the recipient to exchange messages in a language neutral way: for example, the sender can be written in Python and the recipient can be written in Java or C#. Neither side cares how the other side is implemented because they agree on how to interpret the envelope.

REST

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induces desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains architecture to client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol. The following principles encourage RESTful applications to be simple, lightweight, and fast [5]:

- Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
- Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.
- Self descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

- Stateful interactions through hyperlinks: Every interaction with a resource is stateless. That is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

SOA vs. REST services

Web Services are a closer modelling match for more complex applications, since they allow a service to expose an arbitrary interface. They lack structure and are complex to use. REST is certainly simpler because it entails a set of best practices to use resources in a generic way, over HTTP. Although REST provides structure, it supports a single syntax interface and only a set of previously agreed data types. Also, its apparent simplicity hides the functionality that the interlocutors need to be endowed with. In addition, both SOA and REST rely, in practice and in most cases, on HTTP and text based data (XML or JSON).

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 data type (typical in RESTful applications) [5]

The SOA style allows any number of operations for each resource, whereas REST specifies that all resources must have the same set of operations. Real world entities (problem domain) have different functionality and capabilities. REST deals with this in the solution domain by decomposing each entity in basic components, all with the same set of operations, and placing the emphasis on structure. This simplifies access to components, making it uniform, but it also makes the abstraction models of entities harder to understand (hidden by the decomposition into basic components) and increases the semantic gap between the applications entities and REST resources. This is not adequate for all problems, namely those involving complex entities. We want SOAs flexibility in service variability, or the ability to model resources as close as possible to real world entities;

The REST style allows structured resources, whereas SOA does not. Real world entities are usually structured and in many applications their components must be externally accessible. Just using a black box approach (yielding a flat, one level resource structure) limits the options for the architectural solution. We want RESTs flexibility in resource structure [5]. JSON is much simpler than XML and, thanks to that; its popularity has been constantly increasing [6]. The evolution of the dynamic nature of the Web, as shown by JavaScript and HTML5 [7], hints that data description is not enough anymore and distributed programming is a basic requirement. The serialization formats used in 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) and others. However these are compression technologies, which need text parsing after decompression. Recent native binary serialization formats, such as Protocol buffers and Thrift [8], do not have this problem.

Study of SOA and REST

This section explains the most important interoperability concepts related with this project and contains the respective references. Popular current Integration technologies available today are based on the SOA and REST architectural styles. We will see basic implementation of these technologies.

SOA Example

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. Compared to RPC, coupling has been reduced and interoperability increased [9]

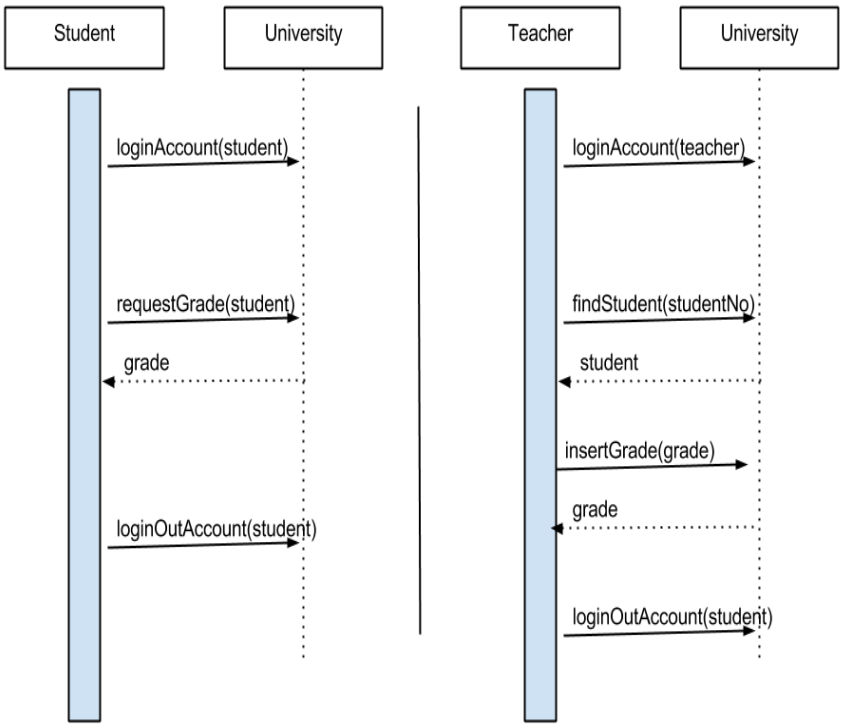

Figure 3 - SOA solution for student and teacher to distribute and show exam results

Figure 3 illustrates the SOA solution for student and teacher to distribute and show exam results by describing the application from the viewpoint of student and teacher. Each of the interacting entities is modelled as a resource, defining a Web Service with operations as needed to support the corresponding functionality.

The SOAP HTTP request uses the HTTP POST method. Although a SOAP payload could be transported using some other method such as an HTTP GET, the HTTP binding defined in the SOAP specification requires the use of the POST method. The POST also specifies the name of the service being accessed.

```
POST /University HTTP/1.0
Host: www.university.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 328
SOAPAction: "http://university.com/grades"

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>
    <m:GetStudentGrade xmlns:m=" http://example.org/grades ">
        <m:studentNo>1234567</m:studentNo>
    </m:GetStudentGrade>

  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 1: SOAP HTTP request to request student grade

The Host: header field specifies the address of the server to which we're sending this request, www.university.com. The next header field, Content-Type:, tells the server that we're sending data using the text/xml media type. All SOAP messages must be sent using text/xml. The content type in the example also specifies that the data is encoded using the UTF-8 character set. The SOAP standard doesn't require any particular encoding. Content-Length: tells the server the character count of the POSTed SOAP XML payload data.

The SOAPAction: header field is required for all SOAP request messages transported using HTTP. It provides some information to the HTTP server in the form of a URI that indicates the intent of the message.

An RPC-style request message usually results in a corresponding HTTP response. Of course, if the server can't get past the information in the HTTP headers, it can reply with an HTTP error of some kind. But assuming that the headers are processed correctly, the system is expected to respond with a SOAP response.

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 359

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>
    <m:GetStudentGradeResponse xmlns:m=" http://example.org/grades ">
        <m:grade>15</m:grade>
    </m:GetStudentGradeResponse >

  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 2: SOAP Response for the request of student grade

The response's HTTP header fields are, for the most part, similar to those of the request. The response code of 200 in the first line of the header indicates that the server was able to process the SOAP XML payload. The Content-Type: and Content-Length: fields have the same meanings as they did in the request message. No other HTTP header fields are needed; the correlation between the request and response is implied by the fact that the HTTP POST is inherently a request/response mechanism. You send the request and get the response as part of a single transaction.

```
package examples.webservices.simple;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
// Standard JWS annotation that specifies that the porType name of the Web
// Service is "SimplePortType", the service name is "Sim-pleService", and the
// targetNamespace used in the generated WSDL is "http:// university.com "
@WebService(name="SimplePortType",serviceName="UniversityService",
            targetNamespace="http://university.com")
// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are document-literal-wrapped.
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
/**
 * This JWS file forms the basis of simple Java-class im-plemented WebLogic
 * Web Service with a single operation: getGrade
 *
 */
public class UnvImpl {
// Standard JWS annotation that specifies that the method should be exposed
// as a public operation.  Because the annotation does not include the
// member-value "operationName", the public name of the operation is the
// same as the method name: sayHello.
 @WebMethod()
 public Grade getGrade (String studentNo) {
 // Code  steps to get grade
 return grade;
 }
}
```

Example 3: Server-side implementation of SOAP with JAX- WS

Next section explains the same operations with REST architecture style.

REST Example

REST is an architectural style that essentially defines a set of best practices of HTTP usage. In the section we will give small example for usage of REST. In REST resources corresponding to actions (such as displaying grade from exam) are created at the server and links to them (such as http://university.com/grades/studentNo) are returned in a response to the client. Following these links has the effect of executing the corresponding action. Such resources are created dynamically, as the state machine process. Following figure shows interaction of student and teacher to distribute exam results.

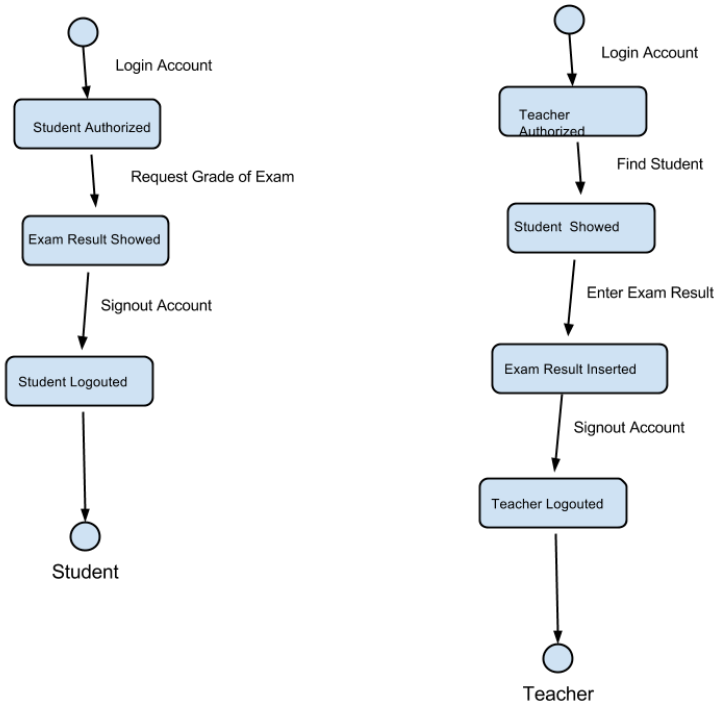

Figure 4 - Interaction of student and teacher to distribute and show exam results

Example 4 shows checking exam result for student in JSON from the university website. We've chosen element names for the JSON representations that are easy for humans to understand.

Performing GET on student's grade URI is very simple. At the HTTP level, it looks like Example 4.

```
GET /Grade/1234567
HTTP/1.1 Host: university.com
```

Example 4 - Requesting grade via GET

If the GET request is successful, the service will respond with a 200 OK status code and a representation of the state of the resource, as shown in Example 5.

```
HTTP/1.1 200 OK
Content-Length: 241
Content-Type: application/json
Date: Wed, 19 Nov 2014 21:48:10 GMT
{
  "id" : 1,
  "name" : "John Smith",
  "grade" : 16,
  "studentNo" : 1234567
}
```

Example 5 – Showing grade with GET

The first line includes a 200 OK status code and a short textual description of the outcome of the response informing us that our GET operation was successful. Two headers follow, which consumers use as hints to help parse the representation in the payload. The Content-Type header informs us that the result is a JSON document, while Content-Length declares the length of the representation in bytes. Finally, the representation is found in the body of the response, which is encoded in JSON in accordance with the Content-Type header. A student can GET a representation many times over without the requests causing the resource to change. Of course, the resource may still change between requests for other reasons. For example, the result of a grade could change by teacher. However, the consumer's GET requests should not cause any of those state changes, lest they violate the widely shared understanding that GET is safe.

The code in Example 6 shows how retrieving an order via GET can be implemented using JAX-RS* in Java.

```
@Path("/order")
public class UniversirtyService
{
 @GET
 @Produces("application/xml")
 @Path("/{studentNo}")
 public String getGrade(
  @PathParam("studentNo")
  Int  studentNo)
 {
   try
   {
      Grade grade = GradeDatabase.getGrade(studentNo);
      if (grade != null)
      {
        // Use an existing XStream instance to create the JSON response
        return xstream.toJSON(grade);
      }else{
        throw new WebApplicationException(Response.Status.NOT_FOUND);
      }
   }catch(Exception e)
   {
      throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
   }
 }
 // Remainder of implementation omitted for brevity
}
```

Example 6 - Server-side implementation of GET with JAX- RS

Inserting a grade at University requires that we POST an order representation in JSON to the service. The create request consists of the POST verb, the ordering service path (relative to the University service's URI), and the HTTP version. In addition, requests usually include a Host header that identifies the receiving host of the server being contacted and an optional port number. Finally, the media type (JSON in this case) and length (in bytes) of the payload is provided to help the service process the request. Example 7 shows a network-level view of a POST request that should result in a newly created order.

```
POST /order HTTP/1.1
Host: university.com
Content-Type: application/json
Content-Length: 239
{
  "id" : 1,
  "name" : "John Smith",
  "grade" : 16,
  "studentNo" : 1234567
}
```

Example 7- Inserting grade via POST

If the POST request succeeds, the server creates an order resource. It then generates an HTTP response with a status code of 201 Created, a Location header contain-

ing the newly created order's URI, and confirmation of the new order's state in the response body, as we can see in Example 8.

```
HTTP/1.1 201 Created
Content-Length: 267
Content-Type: application/json
Date: Wed, 19 Nov 2014 21:45:03 GMT
Location: http://university.com/grade/123467
{
  "id" : 1,
  "name" : "John Smith",
  "grade" : 16,
  "studentNo" : 1234567
}
```

Example 8- Insert response via POST

The Location header that identifies the URI of the newly created order resource is important. Once the client has the URI of its order resource, it can then interact with it via HTTP using GET, PUT, and DELETE.

Now that we have a reasonable strategy for handling order creation, let's see how to put it into practice with a short code sample (see Example 9).

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
{
try{
  Grade  grade = extractGradeFromRequest(request);
  if(grade == null) {
     response.setStatus (HttpervletResponse.SC_BAD_REQUEST);
  }
  else
  {
     String internalStudentId = saveGrade(grade);
     response.setHeader(?Location?,
     computeLocationHeader(request, internalStudentId));
     response.setStatus(HttpServletResponse.SC_CREATED);
 }
 catch(Exception ex)
 {
   response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
 }
}
```

Example 9- A Java servlet implementation for inserting grade

In the next chapter we will propose another approach that tries to solve the problem at its source, by defining a common service and resource model, a small set of common services (core API), an interoperability mechanism based on compliance and conformance and an extensibility mechanism that allows providers to build different clouds, based on this core and with support for interoperability.

Proposed Solution

STRUCTURAL SERVICES

One of the main limitations of current integration technologies, based on either SOA or REST, is that they use the document concept as the foundation, with a data description language as the representation format and schema sharing as the interoperability mechanism. Other factors, such as a connectionless protocol (HTTP) and the lack of native support for binary data and contextual information, are limiting for many applications, although not impeditive. To solve these limitations, we propose an architectural style (Structural Services) with the following main characteristics [10]:

- The main modelling concept is the resource, which can be structured and is able to expose that structure. A resource can be identified by a globally accessible path, starting at a directory root or at some other previously known resource.
- Operations are first class resources but do not expose structure, only behaviour. Data resources can be structured but expose state only (apart from basic getters and setters). Other resources are typically composed of operations and data resources.
- Resources are active and can respond to a message sent to it, by automatically choosing an operation adequate to process that message. If the receiver itself is an operation, that is the one invoked.
- Messages are themselves resources and can include operations.
- Interoperability is based on structural compliance and conformance, not on schema sharing or predefined media types. This means that two interacting resources need to be interoperable only in the operations and/or data actually accessed (partial interoperability) and not on the entire definition of their services. This is a way of reducing coupling
- A dual representation format: text, a distributed programming language, and binary, obtained by compiling the text source and using TLV (Tag, Length and Value) binary markup [18]. Messages can include only source, binary with names and only binary. The latter is the most efficient, but is not self-descriptive. Therefore, the server has the capability of caching mappings between received messages and the parameters of the invoked operations, which the client can use (using a token returned by the server to identify a mapping) if the message is repetitive. The protocol supports retransmissions of messages with self-description information if for some reason the token is not valid.
- A layered protocol, with a generic message level separated from the transport level. There is no requirement for transport reliability. If the application needs it, and the protocol does not implement it, the application needs to do it with timeouts and exception handling.
- A context passing mechanism, backward and forward, achieved by passing two arguments (one functional, another contextual) to and from operations. Exceptions have been unified and included in the backward context mecha-

nism. Support for this is provided by the distributed programming language and the message level protocol.

One of the advantages of Structural Services is the ability to lean more towards SOA or REST, according to modeling convenience, and mix several approaches and patterns in the same problem.

It is important to acknowledge that the REST style requires a fixed set of operations, but without stating which operations should be in that set. The HTTP implementations use some of the HTTP verbs, but this is just one possibility. The Structural Services style allows a further step in flexibility, by opening the possibility of using several fixed set of operations in the same problem. We can have not only both extremes (one set for each resource, SOA style, or one set for all resources, REST style) but also any intermediate combination, with different sets of operations for different sets of resources, according to the polymorphism based on compliance and conformance.

It could be tried to implement the Structural Services style using current technologies. For example, it could be extend WSDL documents (with a structural section to support structured resources) or emulate variable set of operations on top of HTTP verbs. But these technologies and the corresponding set of tools were not conceived for this and compliance and conformance checks would just be too slow. The simplicity of REST would be lost and the complexity of SOA would be increased.

Therefore, it is followed a different approach, by placing basic interoperability at the resource and service description levels, instead of merely at the data level. We defined and implemented a distributed programming language (SIL – Service Implementation Language), with a classic syntax style and an object-oriented look and feel, with the goal of providing native support for both data components and operations, as well as structural interoperability, based on structural compliance and conformance [19]

This is the only way in which we can test the new features of Structural Services without the limitations of current technologies. Obviously, the goal is to provide a proof of concept and not to compete with established technologies or tools.

SIL – Service Implementation Language

SIL is a distributed programming language, in which references are global (such as URIs), not local (pointers) and, unlike typical programming languages, assignments have structural copy semantics. This means that, in an assignment to a structured resource, only the components of the value to assign that comply with the target resource are actually assigned [11]. This is similar to what an XML processor does, by processing only the tags it recognizes and ignoring the rest. [9]

SIL resources and messages are completely self-describing and it is with a schema mechanism that differs from that of usual schema languages in two fundamental ways: One of them is there is no separate schema language. SIL, itself fills this role. This is possible because the schema pertains to one resource only and there are no types, only values with associated variability. The other one is the schema is specific of a given resource (document, message or service implementation), not of a set of

documents. Instead of ensuring compatibility between resources by sharing a common schema, structural interoperability is used to check compatibility whenever needed. This reduces coupling and compatibility. [12]

SIL has source text format for users and a binary format for computer processing, either to be stored in a file or sent in a message. For programming, the source text is the input and the binary format is derived by the compiler. For runtime generated messages, only binary is used, although a decompiler can generate a source text format. The binary format usually includes metadata that supports self-description.

A message in SIL is a resource, just like any other, and can include operations. The message protocol is the simplest possible to allow a message request and reply, independently of message content. All the rest is extensible and uses the envelope approach, in which a message is encapsulated with further control information into another (the envelope) at the sender and retrieved from that envelope at the receiver. Security, for instance, can use this mechanism. There are no specific purpose headers. The message protocol supports asynchronous messages (with futures that can be cancelled if the client gives up waiting or decides otherwise) and application faults (exceptions). Each SIL server maintains a message ID generator, based on a non-repeatable, large sequence pseudorandom number generator. The message ID is used to correlate a response to the original message sender (which may be blocked, waiting for that response, if the message was asynchronous).

Each resource in SIL is represented by a SPID (SIL Public Interface Descriptor), which corresponds to a Web Service's WSDL but much more compact and able to describe both structure and operations. It is automatically obtained from the resource description itself by including only the public parts (public component resources and operation headings). Unlike XML or even JSON, there is no separate schema document to describe a resource.

A compiler for SIL based on ANTLR [22], which is able to produce the SPID automatically and to convert source to instructions and data in a binary format, using TLV as described above. An interpreter then executes the binary code (silcodes, similar to Java's byte codes). The current implementation, in pure Java, is optimized for flexibility rather than performance and is roughly 50 times slower than a Java Virtual Machine (JVM). Much of that time is spent just on method dispatch, the mechanism used to execute the various silcodes. A C/C++ based interpreter would be faster but less portable and harder to develop and to maintain. Given the need for flexibility and control of implementation, we did not use a JVM as a compilation target.

Jetty application server is used to support distribution, but any other server will do. In fact, we only need a handler for the protocol, HTTP or any other. For message exchange, only a transport level protocol is required. SIL's own message protocol is implemented on top of that. it currently uses WebSockets [7], with a cache for automatic connection management, but any lower level message transport protocol will be enough, provided that it implements the level of reliability required by the application or the application provides that itself.

The Jetty server connects to a SIL server (to handle the SIL message level protocol) that hosts a resource directory for service discovery. This is a regular service, just like any other, which contains references to the SPIDs of the resources registered in it.

This directory can be searched for a suitable service by supplying keywords and/or a SPID as required by the client. The directory then searches for these keywords in the registered SPIDs and performs a structural interoperability check to ensure that the returned references to SPIDs are conformant to the SPID used in the search. Similarity ranking [20], as an aid for semi-automatic service search, is not yet supported.

Service composition is done simply by defining a new resource that wraps a set of references to the resources with the existing services and implements the composed behaviour with new operations that invoke the existing services through those references. In essence, this is nothing more that classical object composition, but now with distributed interoperability. Given the support for both behaviour and structure, the composition style can vary, from a more operation-centric to a more data-centric approach.

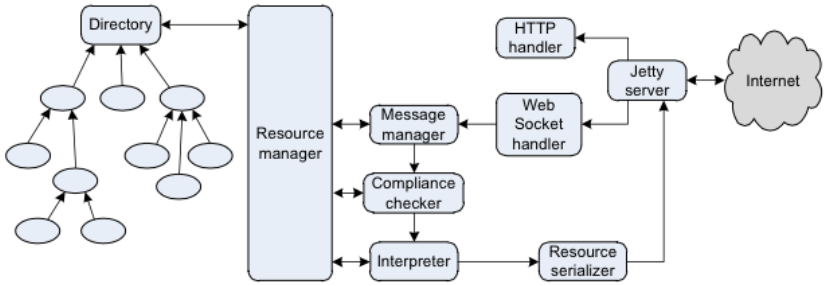

Figure 5 – Basic architecture of a SIL node

The architecture of a SIL node can be described as follows [9]:

1. The application server (Jetty, in this case) is the interface to the network. It supports several message handlers, which means that it can deal with several message level protocols. We have only implemented two, HTPP and the message protocol of SIL. Only the steps ensuring the latter are described here. There is a list of handlers to be invoked and each checks whether it recognizes the message format. The first one to do so gets the message for further processing. All SIL messages begin with "SIL", encoded in UTF-8;
2. A message received is handled next by the Message Manager, which determines the recipient of the message, which streams are present in the message and, for some message types, whether a compliance token is;
3. The Resource Manager implements the access to the structured resources registered in the Directory, obtaining internal references (indices in a resource table) to resources targeted by messages or by distributed references (URIs, for example);
4. The Directory is the root resource and implements several operations, such as searching for a resource which conforms to a given SPID. The

resource tree depicted in Figure 5 shows only containment relationships. Any resource can have a distributed reference to another, but only if it is registered as globally accessible in a Directory. This means that resources can be locally reached from others during execution of a SIL program, but only registered resources, directly in the Directory, can be addressed by a global, distributed reference;

5. The Compliance Checker performs type compliance between the message and the addressed operation or resource. It can do so in text or binary formats, since each has all the information needed, as long as the metadata stream is also present. Naturally, this is faster when done in binary. Messages that include a compliance token, obtained in a previous checking, can skip this step, as mentioned before in the section Serialization Format;
6. If the resource targeted by the message is not an operation, the Compliance Checker must go through the various operations defined in that resource, to find one which the message complies with. A Protocol Fault is returned if none is found;
7. Once the target operation has been identified, a thread is created in the silcode interpreter to execute that operation's code, produced previously by the SIL compiler before the resource has been registered in the Directory. That code can access other resources, according to what has been programmed in the operation;
8. If the operation needs to send a message to another resource, it has to pass the resource to send as a message to the Resource Serializer, which is done by the send operators (<-- or <||, according to whether the message is synchronous or asynchronous, respectively).

This section illustrates the distributed programming language, SIL that developed to implement the Structural Services style.

```
// Teacher
define gradeValue as [0..20]; integer range
define student as {
  lectureId: integer; //ID of lecture
  primitive (-> number); // unnamed operation (get student)
  setGrade: primitive ({&any; number}); //structed parameter

};

//state component
grade: {
  lectureId: integer  // ID of lecture
  primitive (-> gradeValue); //unnamed operation (get Grade)
};
studentArray: array student; //array can grow
//operation componenets

init: operation (array integer){ // array has IDs of Students
  for i (0..in.size)
        addStudent <-- in[i];
};

addStudent: operation (integer) {
  st: student;
  st.studentGrade = in;
  studentArray.add <-- st;
};

getAverageStudentGrade: operation (-> number) {
  total:number; // initialized to 0
  if(studentArray.size == 0)
    reply 0;        //default value
  for i (0..studentArray.size)
     total += studentArray[i].grade;
  reply total/studentArray.size;//compute average
};
```

Program 1- Some features of SIL.

Program 1 illustrates some of features of SIL with a simple description like JSON structured resources are defined between curly brackets and named components with a colon. SIL uses the same mechanism to define named operations (with the key word operation), which can be primitive (implemented in another language, with keyword primitive). Each operation can have only one input and output parameter (but which can be structured), separated by the token ->. These can be accessed in the body of the operation with the predefined identifiers in and out respectively.

Definitions are only auxiliary and generate no components until used. Components can also be declared inline. In this case, an operation without name with matching input parameter will be invoked. This usually corresponds to a GET operation in REST.

The Controller has tree operations (init,addStudent and getAvarageStudentGrade), all non-primitive. This means that they will be executed by the SIL execution platform, in portable way. The compiler transforms their instructions into silcodes (equivalent to Java's bytecode) which are executed by a virtual machine. Resources that

have no primitive operations can be suspended, migrated from one server to another and have executed resumed at the new server.

This resource path, ending in an operation, is typical of the REST style, but in SIL, it blends seamlessly with the SOA style.

```
{
t: number;
g: integer;
t= controller.studentArray[i] <||; //asynchronous message
g= controller.grade >--;           //synchronous message
controller.studentArray[2].setGrade(&mark; 15);

project: operation (integer){
  ...  // deal with project
  };
};
```

Program 2 - SIL Client resource

The Client resource could have a definition such as shown in Program 2. An asynchronous message (with the <|| operator) invokes the unnamed operation of student1 in the Controller's array but returns a future immediately (stored in t), so that processing can proceed concurrently with the message request.

The operation setGrade in student2 is sent a message composed of a reference to the student's grade operation (obtained with the & operator). The first component of the setGrade operation's parameter is declared also as a reference (again, using the & operator), with the predefined value any. All resources comply with any, which means that any resource can be used to receive that grade.

SIL and Microservices

A microservice architecture promotes developing and deploying applications composed of independent, autonomous, modular, self-contained units. Microservices are fine-grained units of execution. They are designed to do one thing very well. Each microservice has exactly one well-known entry point. While this may sound like an attribute of a component, the difference is in the way they are packaged. Microservices are not just code modules or libraries – they contain everything from the operating system, platform, framework, runtime and dependencies, packaged as one unit of execution. Each microservice is an independent, autonomous process with no dependency on other microservices. It doesn't even know or acknowledge the existence of other microservices. Microservices communicate with each other through language and platform-agnostic application programming interfaces (APIs). These APIs are typically exposed as Rest endpoints or can be invoked via lightweight messaging protocols such as RabbitMQ. They are loosely coupled with each other avoiding synchronous and blocking-calls whenever possible. [15]

The solutions SIL (and SOA and REST) with respect to their adequacy for efficiently support microservices. When we compare these technologies for micro-

services architecture, SOA is likely the worst, REST is better and SIL is the best (given its service-orientedness and lightweight), opening the door to distributed web programming based on applications built out of an ecosystem of microservices.

In our thesis we will focus more to microservices and it's relationship and similarities with our solution

Working with SIL in cloud computing platforms

In this section in which we will discuss what we intend to do regarding quantitative assessment of SIL, namely in cloud computing platforms, again by comparison with SOA and REST. After presenting SIL examples in cloud environment and comparing them with other solutions, we believing that SIL is simpler and a better modelling tool than REST but has also the capability of implementing complex applications that characterizes SOA. To able implement these examples and compare with SOA and REST, we will need a simulation because in this way we can simulate cloud platform without having to pay for a real one and simulation tools allow researchers to rapidly evaluate the efficiency, performance and reliability of their new algorithms on a large heterogeneous Cloud infrastructure. The simulation will allow making quantitative evaluation easier and reducing the dependency on the SIL implementation. For simulation we will use CloudSim, which is a new, generalized, and extensible simulation framework that allows seamless modelling, simulation, and experimentation of emerging Cloud computing infrastructures and application services. We will use CloudSim to simulate several situations such as varying number of servers, load measurement and balancing, migration policies, failures. By using CloudSim, researchers and industry-based developers can test the performance of a newly developed application service in a controlled and easy to set-up environment. Based on the evaluation results reported by CloudSim, they can further fine-tune the service performance [14].

Proposed Work

In our thesis we are going to qualitatively compare SOA, REST and SIL, with examples that exercise typical characteristics of distributed applications. This comparison will be done in different application level such as service level, message level and schema level for both Web (fixed servers) and cloud environments.

First of all we will extend and use examples that we described in your previous sections such as teacher and student scenario. These basic examples will be completed and will be implemented in 3 three different styles. For example one of these examples will be typical of SOA such as emphasizing the service functionality, one typical of REST such as emphasizing data structure and another that would require both and so we will have opportunity to compare these solutions. This will allow us to demonstrate the power of SIL. We will implement these examples in different environments such as Web and cloud. In that way we see differences between these two different environment and adaptabilities of these solutions to the environments.

After this implementation, our work will be essentially playing with these technologies and assess it by comparison to SIL. We will start to setting up demo applications and obtain results in fixed web servers and in cloud environments. This will allows us to compare these solutions in different environments and qualitative results for evaluation

In evaluation step, the results will be evaluated qualitatively and quantitatively. So we will compare the execution times with a varying number of servers and we will test situations such as introducing migrations for load balancing and server failures on purpose then we will analyse and discuss the obtained results.

During the implementation of SIL architecture for Web services and cloud environment, my adviser will play important role with understanding difficulties and helping to resolve them easily.

Conclusion

The purpose of SIL is to show that a single language can replace many of existing technologies especially for cloud technologies. The simple description and examples used in this chapter are not enough to fully show how this can be accomplished, but the most fundamental ideas are:

Move from client-server to peer dialog, in a service oriented paradigm; Base interoperability in compliance and conformance, not schema sharing; Derive the schema directly and automatically from the document, instead of having a separate document with different rules; Support and describe both data and code, as well as service and resource architectural styles; Use one serialization format (text based) for people and another (binary based) for computers, but derive the second from the first automatically; Use a simple protocol as the underlying communication mechanism.

To the best of our knowledge, there is no other proposal with such a wide range of objectives, while maintaining the decoupling and distributed interoperability that characterize current Web applications and technologies. The new solution SIL, has been explained in the previous chapter, has been motivation for the design of the alternative solution for current solutions. We can achieve following principles with SIL;

- Programming language style and binary can be used for resource serialization format because using text with markup complex for human reading and hard for machine processing. But programming language style is easier for human reading and binary is perfect for machine processing and they can link with a compiler.
- Text markup does not allow complete separation of data and metadata. But SIL support automatic use of metadata, just it is needed. When schema does not change, send only binary data, with design time checks done by compiler.
- Message Protocol support asynchronous messages and binary data.

- SIL support structural interoperability, which decreases coupling it is not based schema level interoperability on schema sharing like done in XML documents or implied schemas like in JSON data
- SIL supports a variable number of operations for each resource;
- SIL supports multi network resource references, so that heterogeneous networks can be seamlessly integrated. In both SOA and REST, interoperability is achieved by using common data types, either by sharing a schema or by using a previously agreed data type (typical in RESTful applications).

Scheduling of Future Work

For future work, we will start by presenting examples in cloud environment to assess and compare qualitatively the existing solutions and SIL. We will start to show our demonstration as well as guaranteeing the correctness, utility and value of our work. We preview that this task should take around one month to conclude.

After that, we will evaluate the solution proposed with existing solutions in cloud environment by implementing an example in SIL that assumes an elastic computing environment and measuring the execution time with a varying number of computing nodes. We estimate this task will take around one month of work.

Lastly we will analysis the results obtained in the case studies and this task will take 2 weeks of work. We shall finish this phase with the writing of papers for scientific community review. We estimate to write the dissertation around three weeks. This future work plan is exposed in the Gantt chart in Fig. 6

	Task Name	Duration	Start	Finish
1	**Dissertation**	**90 days?**	**Fri 06-02-15**	**Thu 11-06-15**
2	Prensent examples to assess and compare qualitatively the existing solutions and SIL	25 days?	Fri 06-02-15	Thu 12-03-15
3	Evaluate the solution proposed with existing solutions in the cloud enviroment	30 days?	Fri 13-03-15	Thu 23-04-15
4	Analysis the results optained in the case studies	15 days?	Fri 24-04-15	Thu 14-05-15
5	Write Dissertation	20 days?	Fri 15-05-15	Thu 11-06-15

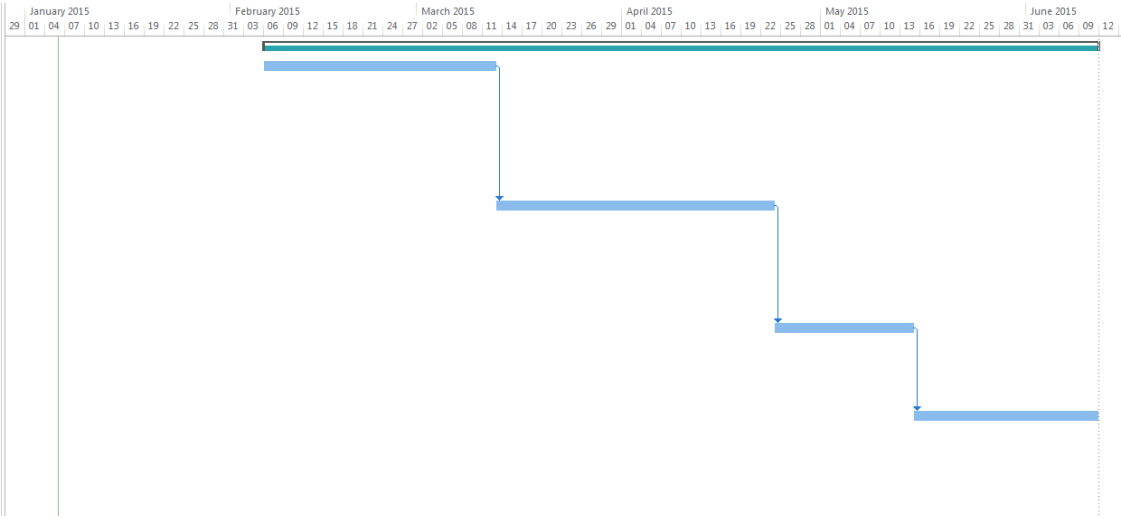

Figure 6 – Future Work Gantt Chart

References

1. Hohpe, G. and Woolf, B.:
   Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston: Pearson Education, Inc. pp 41. (2004).
2. Chris Ferris, Michael Champion and Dave Hollander.:
   Web Services Architecture, W3C Working Group Note 11, February 2004, http://www.w3.org/TR/ws-arch/ [Visited: 18 December 2014]
3. Robert Englander.:
   Java and SOAP, O'Reilly,ISBN:978-0-596-10332-3, pp 2, (2002)
4. Oracle,
   The Java EE 6 Tutorial (January 2013) PartNo: 821–1841–16, http://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf [Visited: 22 December 2014]
5. José C. Delgado.:
   Improving Data and Service Interoperability with Structure, Compliance, Conformance and Context Awareness, Big Data and Internet of Things: A Roadmap for Smart Environments, Studies in Computational Intelligence Volume 546, 2014, pp 35-66, Doi:10.1007/978-3-319-05029-4_2, (2014)
6. Severance, C.:
   Discovering JavaScript Object Notation. IEEE Comp. 45(4), pp 6–8, (2012)
7. Lubbers, P., Albers, B., Salim, F.:
   Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development. Apress, New York, (2010)
8. Sumaray, A., Makki, S.:
   A comparison of data serialization formats for optimal efficiency on a mobile platform. In: Proceedings of 6th International Conference on Ubiquitous Information Management and Communication. doi:10.1145/2184751.2184810, (2012)
9. José C. Delgado.:
   Service Interoperability in the Internet of Things, Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence Studies in Computational Intelligence Volume 460, 2013, pp 51-87, Doi 10.1007/978-3-642-34952-2_3, (2013)
10. José C. Delgado.: Unifying Services and Resources: A Unified Architectural Style, Handbook of Research on Architectural Trends in Service-Driven Computing, DOI: 10.4018/978-1-4666-6178-3.ch016, (2014).

11. José C. Delgado.:
Bridging the SOA and REST architectural styles. In: Ionita A, Litoiu M, Lewis G (eds.) Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments, IGI Global, Hershey PA, (2012)
12. José C. Delgado.:
Structural interoperability as a basis for service adaptability. In: Ortiz G, Cubo J (eds.) Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solutions, IGI Global, Hershey PA,DOI: 10.4018/978-1-4666-2089-6.ch002, (2012)
13. Garg, S.K. and Buyya, R.:
Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on 5-8 Dec. 2011, pp 105 - 113, (2011), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6123487&tag=1 [Visited: 16 December 2014]
14. Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose and Rajkumar Buyya.:
CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software – Practice and Experience, 2010 John Wiley & Sons, Ltd, DOI: 10.1002/spe, pp 41:23–50, (2010)
15. Janakiram MSV.:
Microservices: How to prepare next-generation cloud applications, January 2015
http://www.computerweekly.com/feature/Microservices-How-to-prepare-next-generation-cloud-applications?asrc=EM_ERU_38406846&utm_medium=EM&utm_source=ERU&utm_campaign=20150107_ERU%20Transmission%20for%2001/07/2015%20(UserUniverse:%201293081)_mykareports@techtarget.com&src=5347446
[Visited: 07 January 2015]
16. Tim Bray, Jean Paoli, and C.M Sperberg-McQueen.:
XML Extensible Markup Language (XML) 1.0, 10 November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/ [Visited: 07 January 2015]
17. Charles F Goldfarb and Paul Prescod, XML Handbook 4th Edition, Prentice Hall NJ 07458, pp 653, (2002)
18. Olivier Dubuisson.:
ASN.1 Communication Between Heterogeneous Systems,OSS Nokalva, pp 396, (2000)
19. Raja Ramanathan.:
Handbook of Research on Architectural Trends in Service-Driven Computing, Information Science Reference pp 399, (2014)
20. Formica, A.:

Similarity of XML-schema elements: A structural and information content approach. The Computer Journal,51(2),240-254, doi:10.1093/comjnl/bxm051,(2008)
21. Fielding, Roy Thomas.:
Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, (2000)
22. Terence Parr.:
The Definitive ANTLR 4 Reference, Pragmatic Bookshelf, (2013)