Chapter 1

Introduction

Replication is a mechanism often used to improve response time in many fields such as: Operating Systems (OS), File Systems, and Database Management Systems (DBMS). Replication could be done for data like it is used mainly in Database Management Systems and also for functionality like it is used in distributed operating systems. Many Software manufacturers provide the ability to replicate Web Services [8]. Regardless how it is used, the main goal of replication is to achieve better response time.

Even though replication could improve the response time of a given system, it would be rendered useless if it is not combined with an allocation algorithm to assign functionality or data to replicas. However, using replicas in system introduces an extra overhead for maintenance and data consistency. That is why it is important to combine the replication with an efficient allocation mechanism [8,7,6]. Otherwise, introducing replicas in systems would render the benefit from replication useless when compared to the gained improvement in the response time.

Web Services are becoming a research and industry de-facto for providing functionality in a distributed manner that is usable by heterogeneous environments. Simply put, Web Services are packaged functionality that relies on a set of standards that facilitate the definition of the methods of the Web Services, its number of inputs and formats, and its output numbers and formats. I provide more information about Web Services and its standards in chapter 2. Like any other fields mentioned above, Web Services utilize the concept of replication to achieve better response time and quality of service (QoS). Many big organizations, like Google and Microsoft, provide replications of their published Web Services.

It is practical to use Web services as building blocks for an Internet Database Management System (IDBMS). As legacy DBMS execute queries against local databases using a set of DB operators, using IDBMS, we could execute queries against distributed databases (exposed as Web services) using a set of Web services that compose DB operators' functionality. Using this paradigm, clients issue queries formulated in XML as “Plans”, which is similar to a normal SQL query. The execution of those queries will be facilitated using Web services that could be either A) autonomous Web services which access a Database, or B) Special operators implemented as Web services which implement the SQL operators (i.e. SELECT, JOIN,..etc).

1.1 Research Questions

The utilization of replication provide many performance improvements, however, there are many questions that must be answered in regards to how to improve the replicas. I present these questions briefly below, but I will discuss them in more details in chapter 6

1.1.1 What Replica Allocation algorithm to use?

This is the most asked question which could make or break the replication. Given a set of replicas for a certain Web Service and a set of request with a specific arrival rate, how do you dispatch requests to replicas resulting in the optimum situation (which is best response time)? This question is easier said than answered, since failure to properly address this question could result in two undesired scenarios:

a) Hotspots formulation: which is the case that results from allocating requests to a set of Web Service replicas making them overloaded and hence degrading their response time, and results in queue formation.
b) Idle replicas: where replicas of Web Services are not allocated to any, or few request which makes them idle or underutilized most of the time while other replicas are overloaded with requests.

This question is considered to be the focus of this thesis. It is not a trivial question since replication is done in different instances with different hardware and network configurations.

In this thesis, I will present the Least Response Time (LRT) protocol and show how it is used in a Web Services execution framework and provide experimental results showing that it is superior to other allocation algorithms.

1.1.2 Number of replicas?

Nowadays, the cost of processing power (CPUs) and storage is considered to be cheap, which encourages replication, but at a certain point, introducing more replicas provide no improvement in the response time. In some cases, the improvement of the response time by the introduction of more replicas is not justified when compared to the added overhead of maintaining replicas and data consistency. A threshold must be defined to create a balance that achieves the maximum improvement for the response time with the minimum, possible, overhead to maintain replicas.

1.1.3 What to replicate ?

Functionality included in Web Services usually requires access to a database of some kind to provide the desired results (e.g. Google’s spell checking Web Service). If the functionality alone is replicated (i.e. just the code) without replicating the database, this could render the replication useless since that database management system becomes a bottleneck for all replicas. However, this is not true in all cases, where the Web Services are autonomous and provide computations that require no, or little, access to databases. So, decision makers must decide whether to replicate the code, databases, or both.

Another related question is whether to replicate the code with the database together or to replicate each separately. Some may decide to rely on the database replication algorithms and strategies already implemented on third party Database Management Systems (e.g. Microsoft’s MS Sql Server, and Orcale 10g)

1.2 Research Methodology

The methodology of this thesis is done by doing a survey on the available research material related to the thesis statement below. I formed a hypothesis and investigated many algorithms, did experimental study to choose the most efficient one. The discussion provided in the rest of the thesis provides both analytical and experimental discussions that explore many aspects of the chosen algorithm. The chosen allocation algorithm (Least Response Time) is then included in an execution paradigm (Proteus, see chapter 5) which is used to investigate further aspects of the algorithm. Findings from these experiments are provided as lesson and explained in the conclusion chapter.

1.3 Thesis statement

Based on the questions raised above, I could summarize the contribution of my thesis in the following statement:

“Given a certain amount of Web Service replicas, and given a query formulated as an XML plan how execute the query and direct requests to Web services replicas included in the plan to achieve a better response time“

1.4 Thesis structure

The structure of thesis is defined as follows: in chapter 2, I provide the necessary background, which includes all required definitions of standards and technologies related to the thesis. Chapter 3 will provide a survey of the related work that relates to the scope of the thesis. Chapter 4 provides a discussion of allocation algorithms used in the field of replication along with some implementation algorithms. In chapter 5, I provide an experimental framework to evaluate the allocation algorithms. The main focus will be the Broker element which implements the selected allocation algorithm. Chapter 6 provides the lessons and finding from the experimental results obtained in chapter 5. Finally, in chapter 7 I provide the conclusion and future research areas that could improve the output of this thesis.

Chapter 2

Background

XML Web services are the fundamental building blocks in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where XML Web services are becoming the platform for application integration. Applications are constructed using multiple XML Web services from various sources that work together regardless of where they reside geographically, the programming language used to implement them, or the operating system they run at.

There are as many definitions of XML Web Service, Some of the definitions are:

“A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards” [20]

“A Standardized way of integrating Web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available and UDDI is used for listing what services are available. Used primarily as a means for businesses to communicate with each other and with clients” [18]

“Web services are software components that employ one or more of three technologies -- SOAP, WSDL and UDDI -- to perform distributed computing. Use of any of the basic technologies constitutes Web services. Use of all of them is not required” [19]

“A Web Service is a software component that is described via WSDL and is capable of being accessed via standard network protocols such as but not limited to SOAP over HTTP” [11]

Regardless of the exact definition, almost all definitions have these things in common:

- XML Web Services expose useful functionality to Web users through a standard Web protocol. In most cases, the protocol used is SOAP.
- XML Web services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document.

- XML Web services are registered so that potential users can find them easily. This is done with Universal Discovery Description and Integration (UDDI).

Diverse applications publish the functionalities of their databases and computations as Web Services. A Web Service is a network enabled application component with service-oriented architecture using standard interface description languages and communication protocols that facilitate easy development and deployment of data intensive applications. They use standard XML representations to describe their inputs, outputs, and available operations. For example, one may use the Google WS to invoke operations such as: (1) doGoogleSearch with an input keyword to retrieve the result of an Internet search, and (2) doSpellSuggestion with an input word to retrieve the correct spelling(s). Organizations such as the United States National Institutes of Health (NIH) have and continue to publish the functionality of their data as WSs, e.g., NCBI's WS with operations such as eSearch.

Web services could be used as building blocks for an Internet Database Management System (IDBMS). Using IDBMS, clients could issue queries using what is called XML Plans, which is similar to a normal SQL query, and the execution of those queries will be carried using Web Services that could be either A) autonomous Web Services which access a Database, or B) Special

Operators implemented as Web Services which implement the SQL operators (i.e. SELECT, JOIN, Etc.).

Web Services rely on different technologies and protocols. A Web Service may expose many functions (sometimes they are called methods). These functions along with their inputs and outputs are described using Web Service Definition Language (WSDL) [36]. Clients invoke a Web Service's function by passing an XML message using Simple Object Access Protocol (SOAP) [37]. SOAP is a protocol specification for exchanging structured information between the Web Service and the invoking client. A client could either learn about the Web Service from its owner or through a Web Service directory that is usually called Universal Description, Discovery and Integration (UDDI)[31,32]. Due to their popularity and standardization, Web services become an essential part of the Plan Execution paradigms, which in turn required the introduction of what is called WS-Standards (usually referred to as WS-*). WS-* is collection of specifications allowing Web Services to interact with each other and route information between themselves in a secure and reliable manner.

In this chapter, I will give a brief overview of the technologies, protocols, and standards mentioned above.

2.1 WSDL

The Web Services Description Language (WSDL) is an XML-based language that provides a model for describing Web services.

The WSDL defines services as collections of network endpoints, or ports. The WSDL specification provides an XML format for documents for this purpose. The abstract definition of ports and messages are separated from their concrete use or instance, allowing the reuse of these definitions. A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Messages are abstract descriptions of the data being exchanged, and port types are abstract collections of supported operations (like the example of Google's doGoogleSearch). The concrete protocol and data format specifications for a particular port type constitutes a reusable binding, where the operations and messages are then bound to a concrete network protocol and message format. In other words, we could say that WSDL describes the public interface to the Web Service. Listing 1 below shows a sample WSDL

```
<?xml version="1.0"?>

<definitions name="StockQuote"

          targetNamespace="http://example.com/stockquote.wsdl"

          xmlns:tns="http://example.com/stockquote.wsdl"

          xmlns:xsd1="http://example.com/stockquote.xsd"

          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

          xmlns="http://schemas.xmlsoap.org/wsdl/">

 <types>
```

```
  <schema
targetNamespace="http://example.com/stockquote.xsd"

          xmlns="http://www.w3.org/2000/10/XMLSchema">

    <element name="TradePriceRequest">

      <complexType>

        <all>

          <element name="tickerSymbol" type="string"/>

        </all>

      </complexType>

    </element>

    <element name="TradePrice">

       <complexType>

         <all>

           <element name="price" type="float"/>

         </all>

       </complexType>

    </element>

  </schema>

 </types>
 <message name="GetLastTradePriceInput">

   <part name="body" element="xsd1:TradePriceRequest"/>

 </message>

 <message name="GetLastTradePriceOutput">

   <part name="body" element="xsd1:TradePrice"/>

 </message>

 <portType name="StockQuotePortType">

   <operation name="GetLastTradePrice">

     <input message="tns:GetLastTradePriceInput"/>

     <output message="tns:GetLastTradePriceOutput"/>

   </operation>

 </portType>

 <binding                    name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">

   <soap:binding                         style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

   <operation name="GetLastTradePrice">

     <soap:operation
soapAction="http://example.com/GetLastTradePrice"/>
```

```
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
 </binding>
 <service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port                                    name="StockQuotePort"
binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
 </service>
</definitions>
```

Listing 1 Sample WSDL document

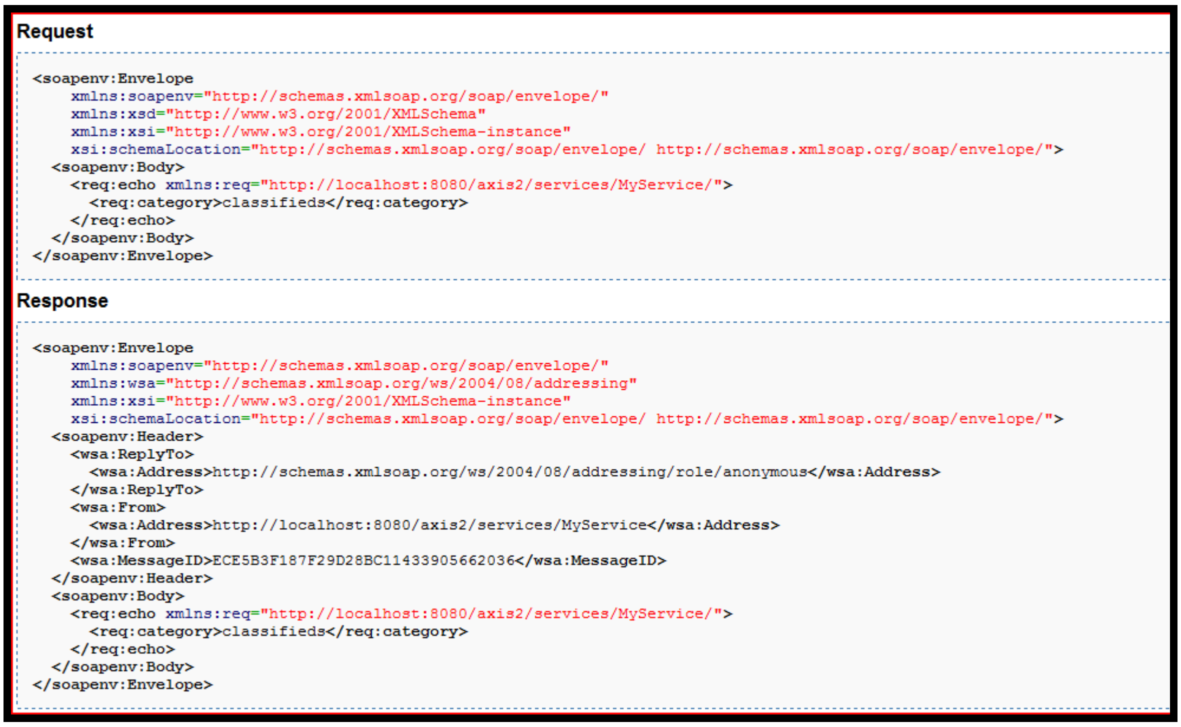
As shown in Listing 1, this Web Service exposes an operation called GetLastTradePrice which receives an input of type String and is called GetLastTradeInput and produces an output of type Float and is called GetLastTradeOutput. As shown in the listing WSDL is designed to be very flexible in both defining data structures and operations (functions). Each Web Service must have a WSDL associated with it; otherwise applications cannot invoke this Web Service properly. WSDL has helped in simplifying the use of Web Services in such a way that new development environments (e.g. Microsoft's Visual Studio) just requires the URL of the WSDL file and it will automatically generate the code to invoke the Web Service for the developer.

2.2 SOAP

SOAP (Simple Object Access Protocol)[37] is a protocol used for exchanging messages between Web Services and clients and/or other Web Services. SOAP is basically a header defined in XML [17]. A SOAP request is passed to the Web Services when invoking one of its methods and in that case it provides the name of the method inside the Web Service along with the names and values of the input parameters. Results of the execution of the Web Services are also sent using SOAP Response messages and in that case they just contain the names and values of the output parameters. Listing 2 below shows an example SOAP request and Response.

SOAP messages are standardized by using XML, so they are transport protocol-independent and Operating System-independent. However, SOAP messages can be transported in many ways. The most used transport method is over

HTTP/HTTPS protocol which is chosen due to its popularity and support in many environments and operating systems. The SOAP message is passed in the body part of the HTTP request.



Request

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/ http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

Response

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/ http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsa:ReplyTo>
      <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:From>
      <wsa:Address>http://localhost:8080/axis2/services/MyService</wsa:Address>
    </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
  </soapenv:Header>
  <soapenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 2 Sample SOAP response and request

2.3 UDDI

Since that WSDL is required for clients to see the functions included in the Web Service, a client must have a way of obtaining the WSDL. This is where UDDI comes in place. Universal Description, Discovery, and Integration (UDDI) [31,32] is a way for Web service authors and creators to publish their products for others to use them.

UDDI could be deployed either privately within a local network, or publicly in the Internet for everyone to use. For example, Microsoft, IBM, and SAP had publicly available UDDI registries but they were closed on 2006. However, several Web Services discovery mechanisms [2,16] are used to enable the discovery and usage of Web Service.

2.4 WS-*

As the Web Services become more popular, people realized a need to define ways to allow Web Services to interact with each other in a more reliable and secure manner. This resulted in the WS-* standards.

There are many standards defined by many role-players in the Web Service industry, but in this thesis, I will focus on the WS-* standards defined by Microsoft [24,27] and I'll show how I utilized them to run the experiments and generate the analytical results. I will show the list of all WS-* standards in this section, but I will elaborated only on the ones related to scope of this thesis. The WS-* standards defined by Microsoft are:

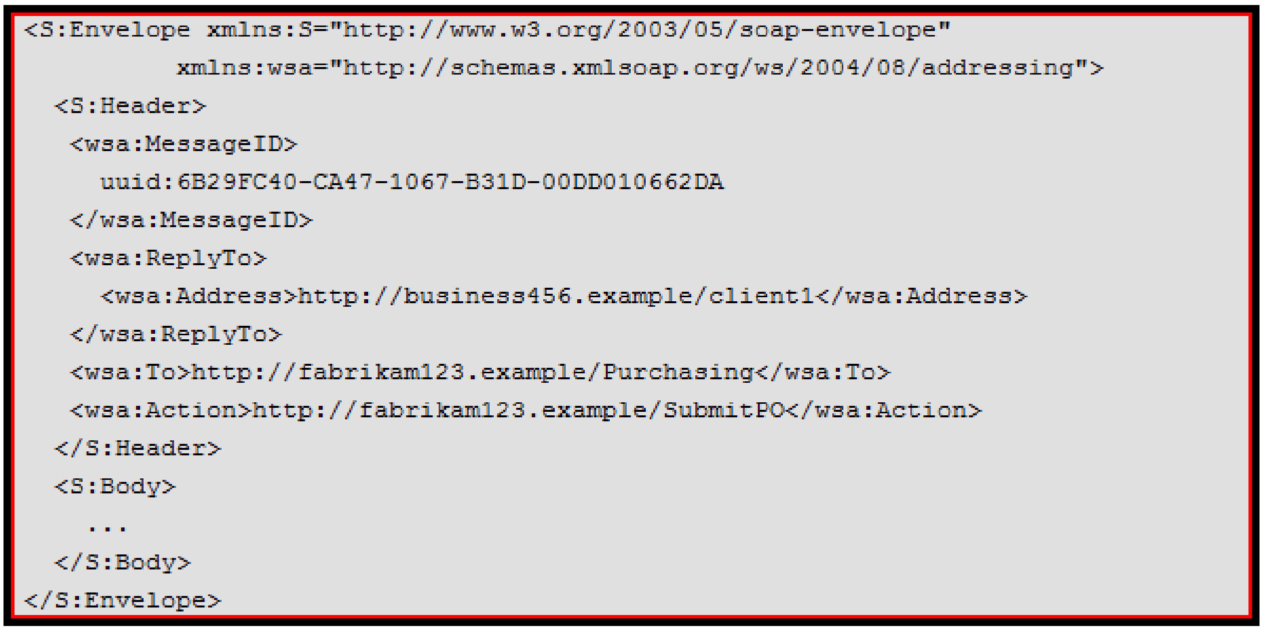
- WS-Addressing
- WS-Enumeration
- WS-Eventing
- WS-Transfer

- WS-Security: SOAP Message Security
- WS-Security: UsernameToken Profile
- WS-Security: X.509 Certificate Token Profile
- WS-SecureConversation
- WS-SecurityPolicy
- WS-Trust
- WS-Federation
- WS-Federation Active Requestor Profile
- WS-Federation Passive Requestor Profile
- WS-Security: Kerberos Binding
- WS-ReliableMessaging
- WS-Coordination
- WS-AtomicTransaction
- WS-BusinessActivity
- WS-Policy
- WS-PolicyAssertions
- WS-PolicyAttachment
- WS-Discovery
- WS-MetadataExchange

- WS-MTOMPolicy
- WS-Management
- WS-Management Catalog
- WS-ResourceTransfer

In the implementation of the suggested execution paradigm, we heavily utilize WS-Addressing[12]. So in the rest of this chapter I will explain how the WS-Addressing works and I'll show how to utilize it in the next chapter.

WS-Addressing (Formerly known as WS-Routing) is a way to facilitate the communication between two (or more) Web Services. It works by applying an additional header to the SOAP message that shows the path of the execution between Web Services. The term EndPoint is used in the WS-Addressing to describe a Web Service that is included in the path of the plan execution. A basic WS-Addressing header should include <was:to> XML element as well as <wsa:ReplyTo> element. A more sophisticated header will also include one or more <wsa:via> XML elements to show intermediate Web Services involved in the plan execution. Listing 3 below shows an example WS-Addressing Header



```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
          xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
  <S:Header>
   <wsa:MessageID>
     uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
   </wsa:MessageID>
   <wsa:ReplyTo>
     <wsa:Address>http://business456.example/client1</wsa:Address>
   </wsa:ReplyTo>
   <wsa:To>http://fabrikam123.example/Purchasing</wsa:To>
   <wsa:Action>http://fabrikam123.example/SubmitPO</wsa:Action>
  </S:Header>
  <S:Body>
    ...
  </S:Body>
</S:Envelope>
```

Listing 3: Sample WS-Addressing Header

WS-Addressing allows a way to define EndPoints and also specifying the functions to be executed in the intermediated Web Services. I will show in the next chapters how to utilize this to facilitate Web Services execution.

Chapter 3

Related Research

In this section I will provide some literature related to the thesis. I will divide the literature in two parts: a) those which talk about the issue of parallelism and how to utilize it to achieve better performance (Most of the literature in this subject is done in the field of Operating Systems and Database Management Systems), and b) those which talk about distributed query execution using Web Services.

3.1 Parallelism and Operating Systems

When talking about parallelism and resource allocation, attention is automatically drawn to the field of Operating Systems. Since the beginning of computers, researchers thrived to get more processing done with better response time. This was achieved by parallelism and efficient resource allocation. Of course, back in the day, computers weren’t as cheap, as powerful, or as fast as they are today. However, as computers advance in terms of speed and capacity, peoples’ and applications demands increase by requiring less and less response time for their required computations.

Even though the subject of this thesis is not related to the field of Operating Systems, I couldn’t write it without mentioning two points that explains the difference between the work that has been done in that field and the focus of this thesis:

- In the field of OS parallelism required is usually done on the same device (server) or, in the case of Distributed Operating Systems, within several devices connected using a very high-speed bus for data transmission [1]. However, when talking about Web Services and the proposed Internet DBMS systems, we must think of devices and servers that are located in different geographical areas with connectivity that is so inferior to the high-speed buses available in servers. This introduces a factor that complicates the techniques used to provide better response time in the field of IDBMS. Network speed and bandwidth did not advance in the same magnitude that CPU, RAM, and storage have, this resulted in Network usually being the bottleneck and the most dominant factor in any response time equation [30].
- The nature of Operating Systems, allows the usage of certain techniques for optimization [28,29], such as Preemptive scheduling where execution of certain jobs could be stopped to allow execution of others. However, in the field of Web Services and IDBMS such techniques cannot be applied. The intuitive explanation is that the Operating System environment is self-controlled and homogenous. However, when dealing with Web Services provided by different organizations, we cannot make such

an assumption because they are usually autonomous and each organization only has control over its own published Web Service.

3.2 Distributed Execution Frameworks using Web Services

Due to the increase of using Web Services as they provide an easy, efficient, and standardized way to expose Databases, there has been several research efforts (including this thesis) to explore how to utilizes these autonomous Web Services and utilize them into a unified execution framework. The research efforts vary from defining a generic programming language for this purpose to actually implementing such frameworks. Below, I will show some of these efforts. The order I present them has no relevance to the importance of the research, nor to its precedence. It is just a personal choice.

3.2.1 Flow-based Infrastructure for Composing Web Service (FICAS)

FICAS [38] was developed in Stanford University. It presents a loosely coupled service-composition paradigm. This paradigm employs a distributed data flow that differs markedly from centralized information flow adopted by current service integration frameworks, such as CORBA, J2EE and SOAP. Distributed data flows support direct data transmission to avoid many performance bottlenecks of centralized processing. In addition, active mediation is used in applications employing multiple Web Services that are not fully compatible in terms of data formats and contents. Active mediation increases the applicability of the services, reduces data communication among the services, and enables the application to control complex computations. FICAS executes queries by separating the control of the query from the dataflow. It also insures the completion of the query execution even if some sources required are not Web service-complaint, by introducing mediators that could handle different sources of data. The figure below shows a sample control flow and data flow in FICAS

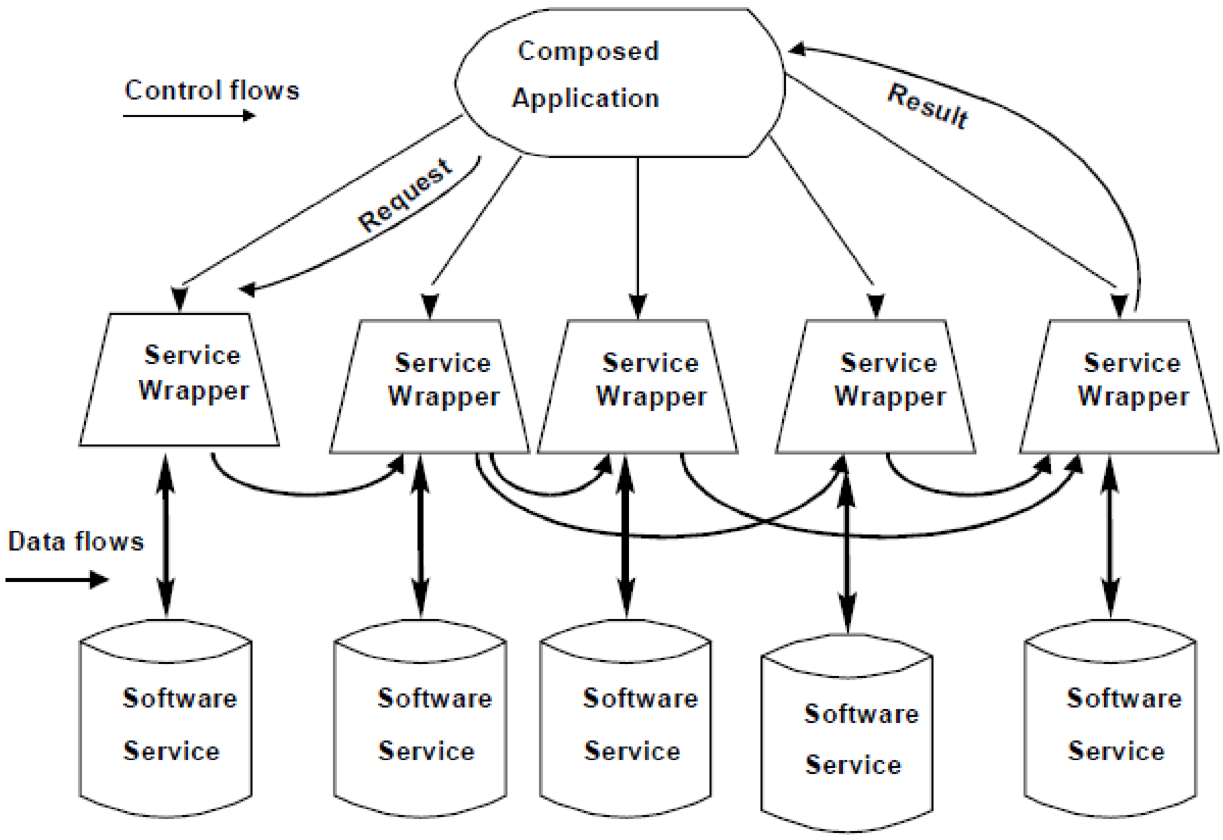


Figure 1: Sample control flow and Data Flow in FICAS

While FICAS defines the framework for the distributed execution, it does not address the issue of response time and how to allocate replicate if they exist.

3.2.2 XL: Platform for Web Services

XL [13] is a programming model that is designed with Web services in mind. It provides programmers with a intuitive model to deal with Web Services in a manner similar to traditional programming languages.

The XL platform consists of two main components a) the XML compiler, and b) the XL virtual machine. The compiler converts the program into a statement graph that is actually a presentation of Web Services and their interaction. This representation is optimized and submitted to the XL virtual machine. Without going into the details of the XL platform, The virtual machine is responsible of invoking the Web Services and collecting the results as the output of the program. Figure 2 below shows the architecture of the XL platform that explains the interaction between its components.

As the authors of the XL platform state “We would like to high-light three important features of the XL virtual machine. First, in order to execute statements in parallel, the virtual machine is multithreaded. Second, the virtual machine is designed to be able to stream the intermediate data between statements; pipelining is a very important feature of our design. Third, in order to achieve scalability and high reliability, the XL virtual machine has been designed to support the migration of processes from one machine to another machine in a cluster (we expect that the platform will be installed on a cluster of servers)". This statement explains the level of parallelism that XL provides. However, it does not explain how is it achieved and it does not explain the allocation process of Web Services replicas if they are introduced in the system.

3.2.3 Business Process Execution Language for Web Services (BPEL4WS)

Business Process Execution Language for Web Services (BPEL4WS)[22] is an OASIS [11] standard that provides a means to formally specify business processes and interaction protocols.

BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces. Figure 3 below shows a how a Web Service is implemented in BPEL4WS.

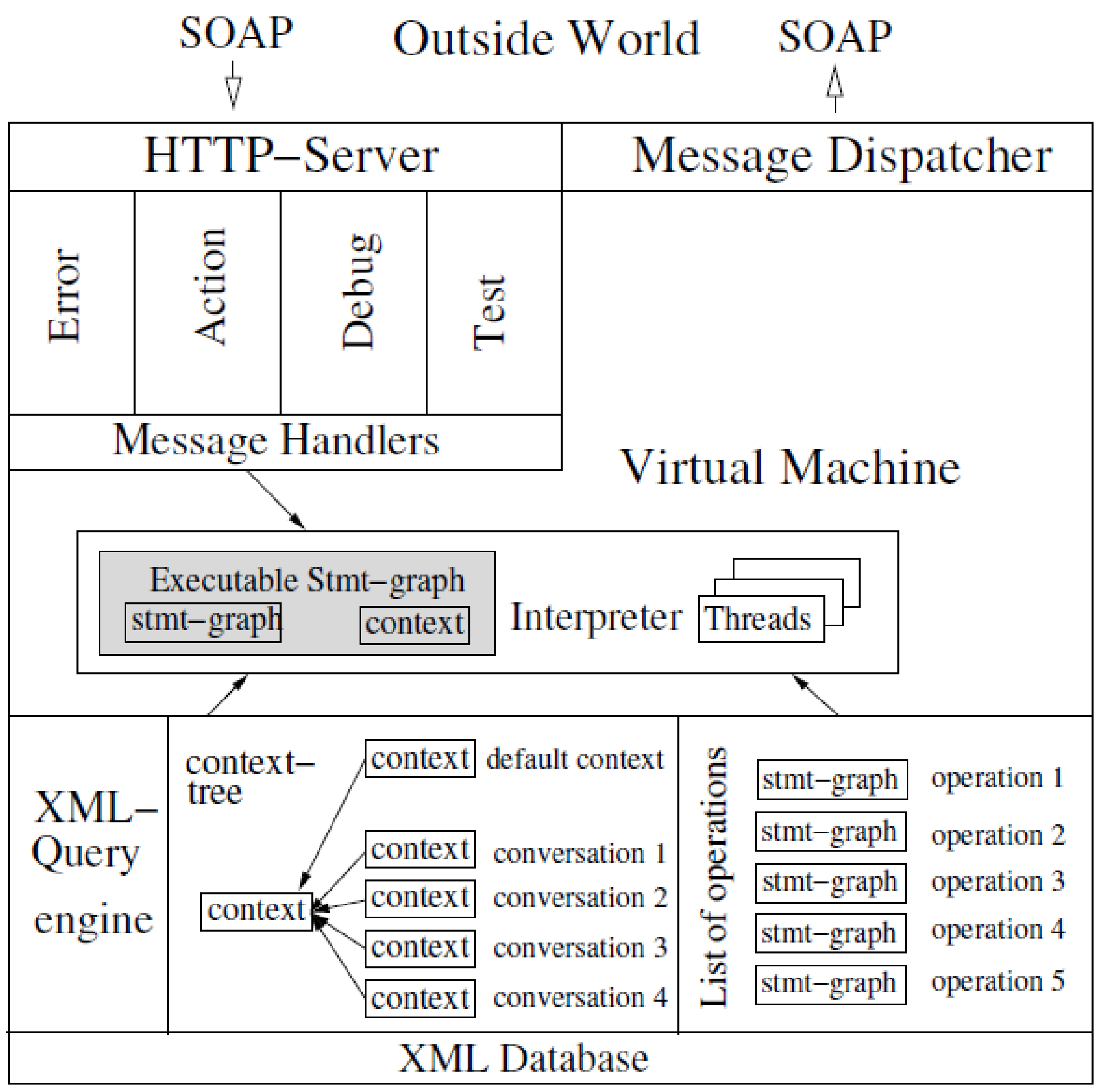


Figure 2: The Architecture of the XL platform

While BPEL4WS describes services composition and enables composition of replicated Web Services, it does not incorporate an algorithm to allocate Web Services on the fly.

Note: Some efforts like [6] are done. They refer to parts of the research leading to this thesis done by myself[7].

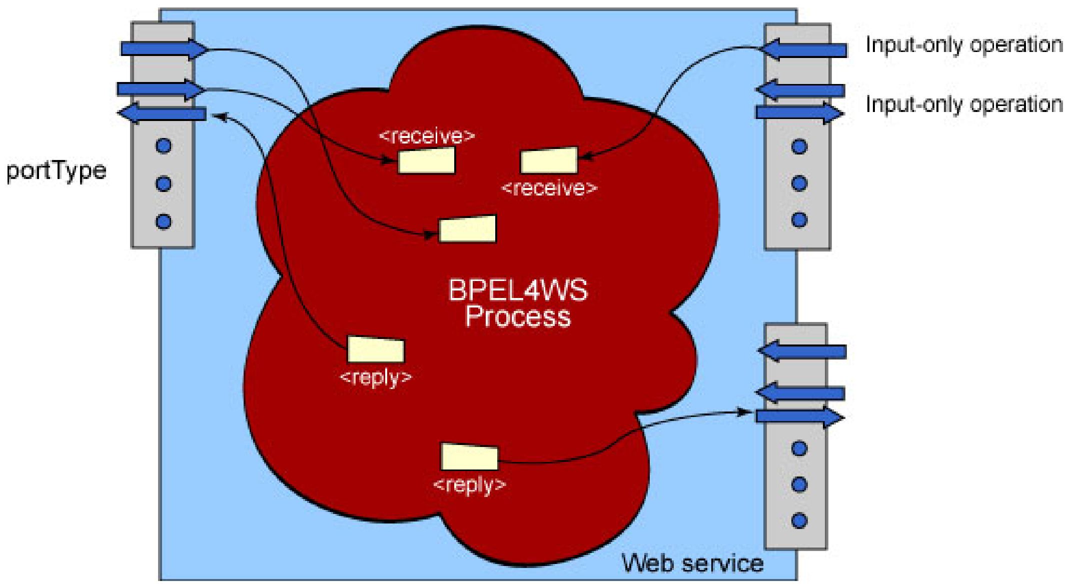


Figure 3: Web Service implemented as BPEL4WS Process

3.2.4 Web Services Flow Language (WSFL)

Web Services Flow Language (WSFL) [14] is an XML Language that enables composition of Web Services. It incorporates two models, namely, Flow Model and Global Model. The former describes the overall composition of Web Services into the execution by specifying the sequence of Web Services along with the data flow between them. The later, however, describes how the composed Web Services would interact with each other, by specifying a set of links with endpoints. Each link defines how a Web Service at one end point would interact with a method in another Web Service at the other end.

As Shown in Figure 4, WSFL is very sophisticated and it handles patterns like loops, conditional loops, branching, joining, and exit condition.

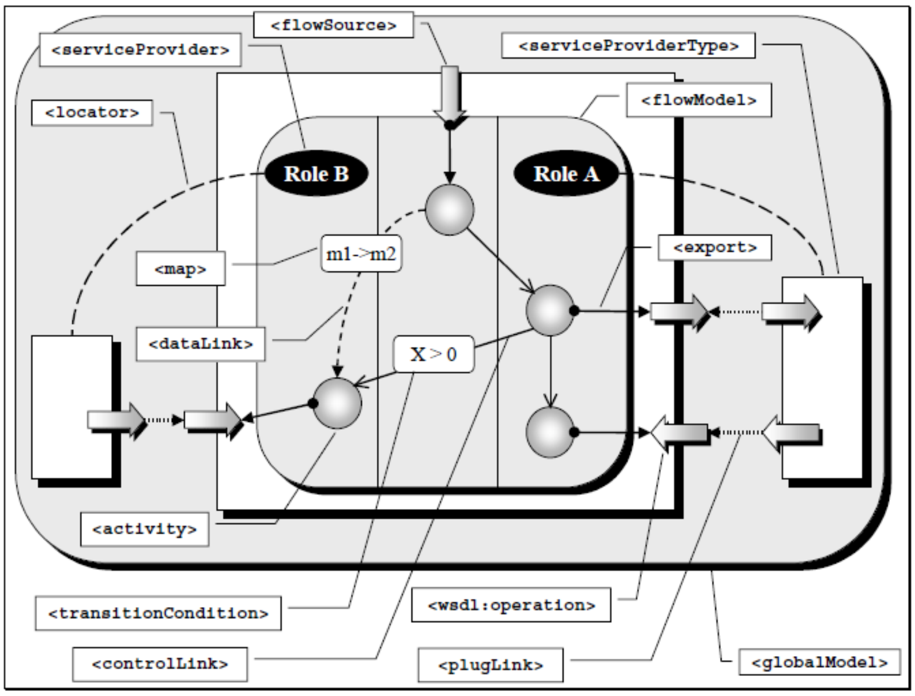


Figure 4:Sample flow in WSFL

Similar to BPEL4WS, WSFL does not include a way to allocate Web Services in the middle of execution. Meaning that Web Services allocation is done before the execution of the plan has started and does not address change in the utilization of allocated Web Services.

Chapter 4

Allocation algorithms: Implementation Alternatives and decisions

In previous chapters, I introduced the idea of utilizing Web Services as building blocks for IDBMS and showed the growing need to do so. In this chapter, I will discuss various allocation algorithms and discuss various implementation algorithms including the Least Response Time Allocation algorithm, which is the main focus on this thesis. The discussion will not present any experimental results as I will show them in the proposed framework described in chapter 5. However, the summary of these discussions will be used and referred to, in the following chapters.

In chapter 3, I showed many research efforts to provide specifications and frameworks for distributed query execution using Web Services as building blocks. All of these frameworks support, in theory using replicated Web Services; however, not all of them provide a mechanism for allocating replicas. In other words, the allocation is done statically and hardcoded in the query plan. It is important to provide a component, a Broker, which sole responsibility is to allocate Web Services. The allocation process must utilize an allocation algorithm that aims for better response time (if not the best response time) which is the main reason why these frameworks are developed in the first place [1,3,4]. The envisioned way of query execution is as follows:

1- The query is formulated as an execution plan. Most of the surveyed frameworks define the execution plan in XML format.

2- The query is submitted to the execution engine. The execution engine is responsible for parsing the plan, identifying Web Services, and controlling the flow of data between them. Due to the way the execution engine works, I will refer to it in the rest of this thesis as the "coordinator".

3- Once the execution of the query is completed, the final results are returned to the coordinator and back to the user/client application.

The figure below shows a generic design of a query execution framework that utilizes a coordinator and broker components

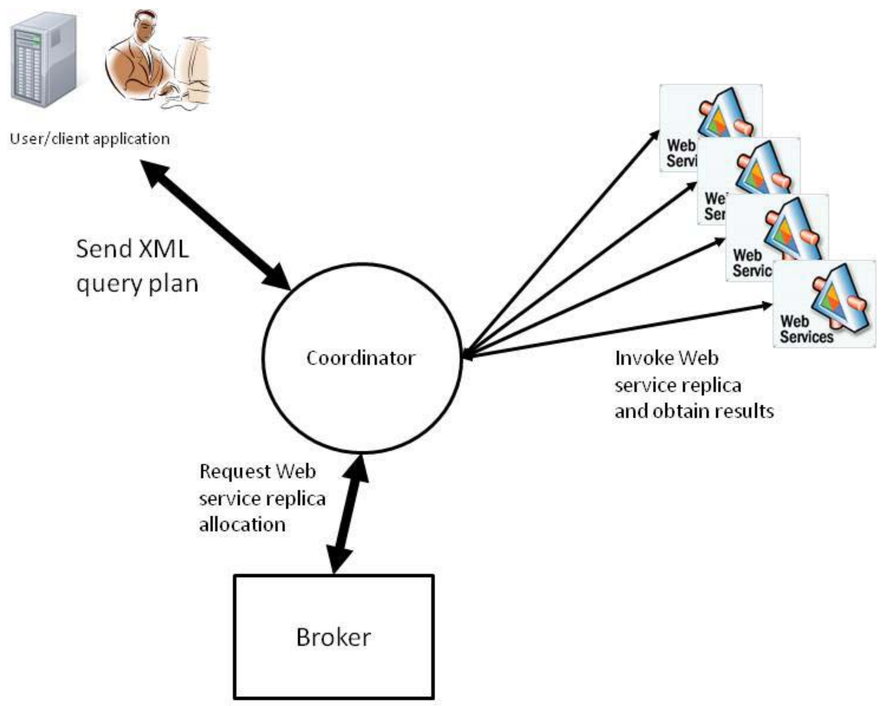

User/client application
Send XML query plan
Coordinator
Web Services
Invoke Web service replica and obtain results
Request Web service replica allocation
Broker

Figure 5: Generic Distributed Execution Framework using Web Services.

The broker is used by the coordinator when a Web Service is referenced. The coordinator will pass the Web Service identifier to the broker which “allocates” a replica of the Web Service and returns its location to the coordinator to use it in the execution of the query plan.

In this chapter, I will assume the existence of the broker component and discuss the following alternatives:

1. Allocation Policy in the Broker
2. Broker Deployment
3. Distributed vs. Centralized Coordinator
4. Intra-Web Services Parallelism vs. No Intra-Web Service Parallelism

5. Collisions

Later in chapter 6 I will show the real implementation of the Broker and Coordinator components and present the experimental results.

4.1 Allocation Policy in the broker

The broker is the brain of the Framework. It tells the coordinator which Web Service replica to use and when to use it. The broker decides which replica of a Web Service to allocate based on an allocation (selection) policy. This policy determines the performance of the whole system. If the allocation policy is faulty, the execution of plans will results in some replicas being over-utilized (hence, slower performance and hotspot formulation) and other replicas sitting idle serving no requests. The Broker maintains metadata about the Web Services and their utilization. This metadata depends on the allocation algorithm and its requirement. I envision the broker to have a database that has the Web service ID as its primary key and a set of replica Profiles (WS_REPLICA_Profile) for each Web Service. The replica profile contains the information utilized by each allocation algorithm. I show this information while we examine the following 4 different allocation policies:

1- Least Recently Used (LRU)
2- Least Recently Allocated (LRA)
3- Least Utilized (LU)

4- Least Response Time (LRT)

Below I present an explanation of each policy.

4.1.1 Least Recently Used (LRU)

The WS_REPLICA_Profile for this policy consists of a time stamp for each copy of a Web Service. This policy allocates the copy of a Web Service which has been least recently used. The time stamp is updated every time a WS finishes executing its currently assigned query. The intuition is to distribute the load between copies of a WS evenly. The pseudo code for the LRU algorithm is as follows:

```
Function LRU (WS_ID) returns WS_URI
{
 WS_CANDIDATES = Lookup (WS_DB, WS_ID) ;
 Sort_Asc (WS_CANDIDATES, TimeStamp);
 // least recently used WS is first on the list

Return WS_CANDIDATES[0].WS_URI;
}
//This method is invoked when a Web Service replica
//finishes execution.
Function Update_TimeStamp (WS_ID, WS_URI)
{
 WS_DB[WS_ID].[WS_URI].TimeStamp = NOW;
}
```

Listing 4: Logic for LRU allocation Algorithm

A limitation of this policy is its ignorance of the assignment of WS replicas to nodes. For example, replicas of two different WSs assigned to the same node might be allocated simultaneously because both were least recently utilized. An obvious improvement is to extend LRU to consider the utilization of participating node.

4.1.2 Least Recently Allocated (LRA)

This is similar to the LRU policy, except that the WS_REPLICA_Profile consists of a time stamp that is updated when the operator is allocated, i.e. before starting the execution. Similar to LRU, LRA does not consider the utilization or service time of the node and suffers from both inter and intra-WS collisions. It can also be extended to consider utilization of nodes.

```
Function LRA (WS_ID) returns WS_URI
{
 WS_CANDIDATES = Lookup (WS_DB, WS_ID) ;
 Sort_Asc (WS_CANDIDATES, TimeStamp);
 // least recently assigned  WS is first on the list
 Update_TimeStamp (WS_CANDIDATES[0].TimeStamp, NOW);
Return WS_CANDIDATES[0].WS_URI;
}
```

Listing 5: Logic for LRA allocation Algorithm

4.1.3 Least utilized (LU)

This policy allocates WSs based on the utilization of their node. It chooses the WS hosted on the least utilized node. Different plans have different input sizes impacting the utilization of each node, which is considered by this policy. The Performance Metadata for this policy includes the utilization of nodes where copies of a WS are hosted. It is the responsibility of each node to update its utilization. We envision two ways to do this, namely, updating the utilization either (1) periodically or (2) updating it when processing of a request has commenced and completed (these two events specify node utilization).

```
Function LU (WS_ID) returns WS_URI
{
 WS_CANDIDATES = Lookup (WS_DB, WS_ID) ;
 Sort_Asc (WS_CANDIDATES, Utilization);
 // least utilized WS is first on the list
Return WS_CANDIDATES[0].WS_URI;
}

Function Update (WS_ID, WS_NODE,New_Utilization)
{
WS_DB[WS_ID].[WS_NODE].Utilization = New_Utilization;
}
```

Listing 6: Logic for LU allocation algorithm

One may combine LU with either LRU or LRA. These policies are in synergy because LU considers node usage while the other two consider WS usage.

4.1.4 Least Response Time (LRT)

This policy estimates the expected service time for each WS and chooses the WS with the Least Response Time. We define response time as the elapsed time from allocating the WS until the WS execution finishes. It is impacted by the hardware speed of the node hosting the WS (service time), and the number of requests waiting in the queue of a referenced WS (queuing delays). The broker in this allocation algorithm maintains Performance Metadata for each replica. It maintains two important values, T_{busy} and T_{st}. The Former indicates the estimated time the Web service replica will be busy till. The later estimates a service time for a single invocation of the Web Service (I explain how these are estimated in the next Chapter). In this allocation algorithm, when the broker receives an allocation request it allocate the Web Service replica with the least response time (i.e smallest T_{busy} value).

```
Function LRA (WS_ID) returns WS_URI
{
 WS_CANDIDATES = Lookup (WS_DB, WS_ID) ;
//Sort, so the least response time WS is first on the list
Sort_Asc (WS_CANDIDATES, Tbusy);
 //updating the Tbusy time for the selected replica
WS_CANDIDATES[0]. $T_{busy}$  += WS_CANDIDATES[0]. $T_{st}$


Return WS_CANDIDATES[0].WS_URI;
}
```

Listing 7: logic of LRT allocation Algorithm

This policy updates the Performance Metadata the same way as LU. However, due to the nature of information stored in the Performance metadata, this algorithm requires an accurate estimation to work properly. I discuss these issues in the next chapter, when we talk about the execution framework.

4.2 Broker Deployment

The broker acts as a dispatcher for all client requests. The broker implements an allocation policy to assign a client request to a Web Service replica. The broker implements an allocation algorithm to enable selection of Web Service replicas.

One could envision three alternative deployments for the broker. Two of these deployments are client neutral because they assume the client is unaware of the broker's presence. The third deployment assumes the client is aware of the broker. I describe these deployments in turn and compare their advantages and disadvantages. All three deployments strive to evenly distribute workload across WS replicas.

4.2.1 Two-Way Transparent Broker (TWTB) deployment

This deployment relies on existing networking standards to direct all requests to the broker. When a client issues a DNS lookup to get the WS's IP address, the DNS server returns the IP address of the broker. The broker receives the WS invocation (Step 1 in Figure 6), parses the HTTP header to obtain the name of the operation requested and then forwards it to the replica selected by allocation algorithm (Step 2 in Figure 6). The broker modifies the packet headers setting the from-IP to be the IP of the broker and the to-IP to be the IP of the selected replica. Once the replica processes the request, the results are sent back to the broker (Step 3 in Figure 6), which forwards it back to the client (Step 4 in Figure 6). The broker maintains translation tables, similar to NAT tables, which map clients to WS replicas, allowing the broker to remain transparent.

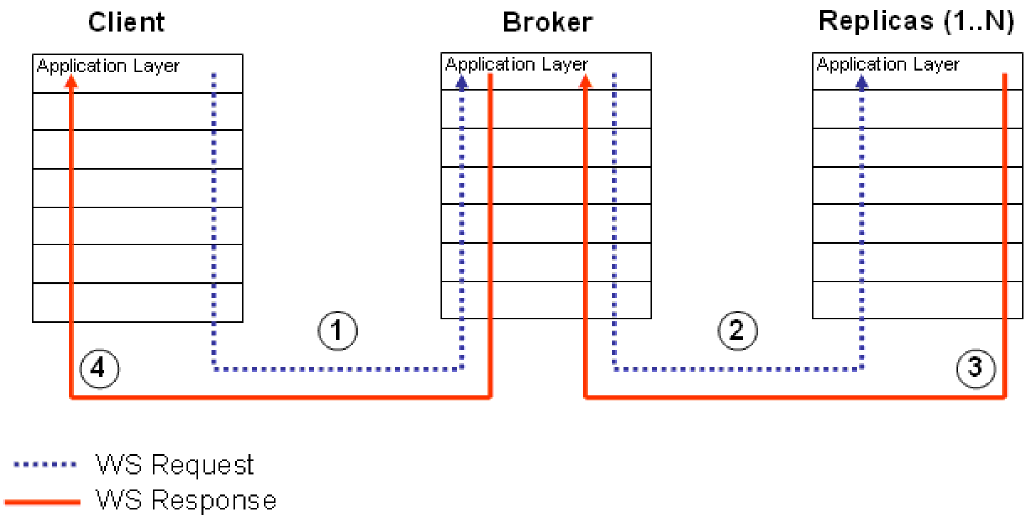


Figure 6: TWTB deployment

In this deployment, both requests and responses to and from the WS are intercepted and processed by the broker. This enables the broker to submit the next request when its previously issued request is processed, eliminating collisions (See section 4.4 below) and providing the broker with precise service time (T_{ST}) profile of each WS replica.

4.2.2 One-Way Transparent Broker (OWTB) deployment

With this deployment, the client obtains the IP address of the broker in the same way as in the previous deployment. The key difference is that the broker only modifies the packet's to-IP to a Web Service replica's IP. When the assigned WS replica completes processing of the request, it forwards its response to the client directly without contacting the broker (Step 3 in Figure 7). This minimizes the load in the broker. At the same time, the broker has imprecise estimate of a replica's T_{ST}. Instead it must estimate the service time of a Web Service replica. One may envision a scenario where Web Service replicas provide the broker with their profiles. Regardless, this deployment may result in an uneven distribution of the load across Web Service replicas.

4.2.3 Broker-Aware deployment

In this deployment clients invoke a WS implemented in the broker with: 1) the name of the desired operation to be invoked (e.g. doGoogleSearch), and 2) the number of client's invocations, required for that service. The broker responds with a list of WS replica URIs and the number of invocations that should be assigned to each replica. Example clients that benefit from this deployment are execution frameworks like Proteus [21] and SANGAM[33] where the coordinator knows the number of requests in advance.

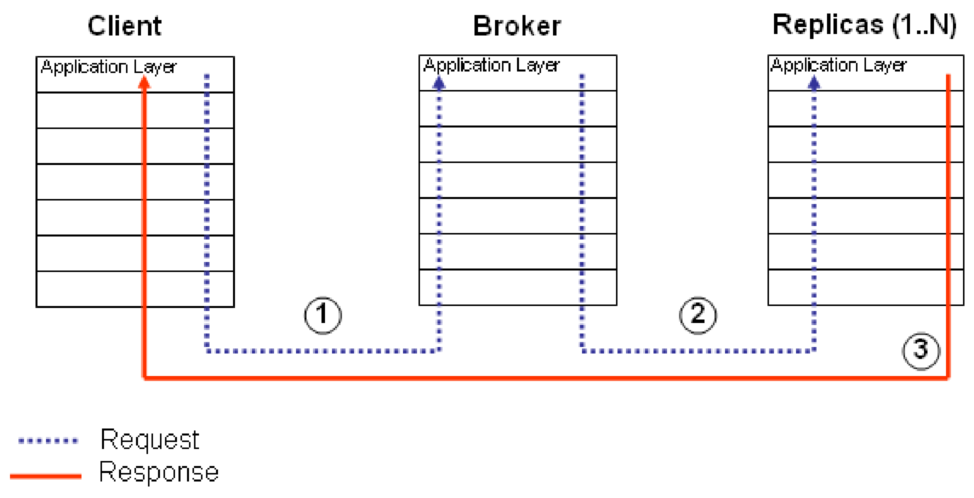


Figure 7: OWTB deployment

The control of when requests are sent to replicas is pushed to the client. In this deployment, the broker has no way to measure the T_{ST} of different replicas, since neither the request nor the response (Steps 3 & 4 in Figure 8) passes through the broker.

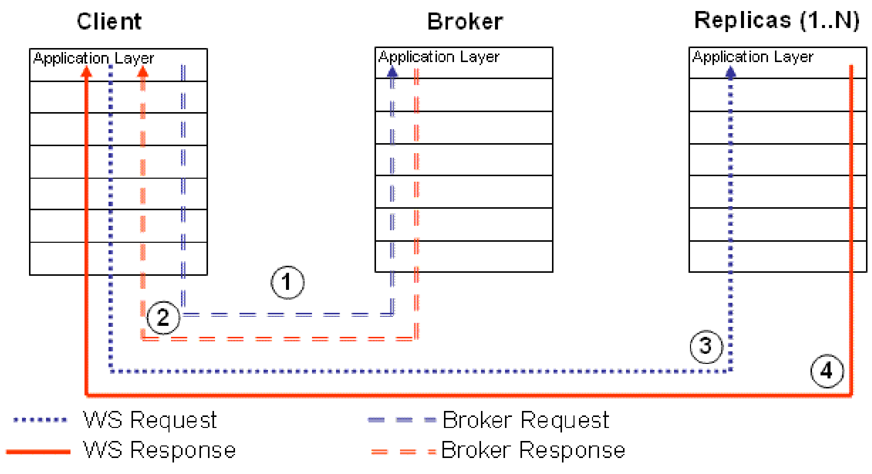


Figure 8: Broker-aware deployment

4.3 Distributed vs. Centralized Coordinator

The coordinator executes plans by utilizing both the autonomous Web Services and execution framework operators. One could envision two types of Coordinators:

- Centralized Coordinator
- Distributed Coordinator

Both types handle plans and autonomous Web Services in the same manner. The top element from the plan is removed then execute, with the results of the execution and the remaining of the plan are passed on to the next Web Service.

The different between the centralized coordinator and distributed coordinator is the way framework Operators (See Chapter 5.4) are implemented. In the centralized coordinator, they are implemented as part of the coordinator's code (as a linked DLL libraries), which means that they must be executed from within the same machine. In the distributed coordinator, framework Operators are implemented as Web Services, which means that one could deploy them in different nodes, allowing for the load to be distributed among different nodes. In addition, in the case of the distributed coordinator, one could deploy more than one replica of the framework operator Web Services, and register them with the broker. This allows the coordinator to choose the operator replica that provides better performance.

This issue is not covered in depth in this thesis, but I believe it is worth investigating to find the trade-off between deploying centralized and distributed versions of the Coordinator.

I could provide hypothetical argument supporting both types of the coordinator, but only experimental results will show exactly what to expect from each. For example, one could argue that the centralized coordinator doesn’t suffer for the extra overhead of calling the broker and requesting the replica of the Proteus Operator and then the network overhead to pass the input relations and wait for the results to come back over the network. But using the same line of thought, one might argue that even though the centralized version of the broker doesn’t have the network overhead, it suffers from the limited extensibility. Meaning that when many plans are being handled by the coordinator at the same time, the CPU will become the bottleneck the performance will degrade.

Again, I don’t have any supporting data to show which type of coordinator is better and what circumstances it should be used in, but the subject itself had to be mentioned.

4.3 Intra-Web Service Parallelism (IWSP) vs. No Intra-Web Service Parallelism (No-IWSP)

It is common for Web services execution frameworks to have many invocations for the same Web Service. For example, assume a case where you have a query plan that queries the Yellow pages directory for certain business time (e.g. restaurants) and then requires a spell check of the restaurant's name. Executing such query, requires the framework to invoke the Yellow Pages Web Service to get a number of records for the resulting restaurants. Then is must issue a request against Google's doSpellSuggestion method for each one of them. For this operation, frameworks require certain kind of operators that performs such invocations and collect the results in one set to be passed to the next operator. In the proposed execution framework (presented in chapter 5) we will call this operator the "Iterator".

The issue of IWSP and No-IWSP must be discussed when designing the Iterator Operator. The Iterator operator is used to send a set of records to autonomous WSs one record at a time. This is because an autonomous WS may accept only one XML element (instead of a set) as input. Iterator invokes Web Services in two different ways: without Intra-Web Service parallelism (No-IWSP), and with Intra-Web Service Parallelism (IWSP). No-IWSP invokes Web Services one record at a time, one replica at a time, resulting in a completely sequential execution. IWSP, in the other hand, allows the execution engine to spawn a thread for each replica of the WS, and each thread invokes a WS replica with records sequentially

To motivate the need for a broker and a Web Service allocation policy, consider the Google Web Service that provides, among others, two operations: (1) Internet search using a keyword, doGoogleSearch, and (2) spell check of words, doSpellSuggestion. Assume there are M replicas of this Web Service with each replica supporting both operations. To simplify the discussion, assume only a single query is executing in the system. This might be a simple query that spell-checks N string tokens. The system may process this plan in two possible ways, with and without intra-WS parallelism. With the later, no intra-WS parallelism (no IWSP), the query plan submits each of its N tokens for processing one at a time. This is the simplest mode of operation that does not benefit from multiple replicas of a WS.

With intra-WS parallelism (IWSP) and assuming N is larger than M, the system submits M tokens (out of N) to M different Google Web Service replicas for processing simultaneously. This is appropriate when the service time of a Web Service replica degrades due to multi-threading caused by submitting multiple tokens to a WS replica simultaneously. To simplify discussion, the term node refers to a single processor PC (or workstation) configured with multiple mass storage devices and a networking card.

We motivate the need for an allocation policy by considering the scenario where the M WS replicas reside on a heterogeneous collection of nodes, providing a different service time. In this case, one node finishes before another. To enhance response time, the allocation policy should assign the M+1st request to the first node that finishes processing its request, keeping all nodes busy until all N requests have been processed.

I will show experimental results in the following chapters that explain the trade-offs and show the differences between IWSP and No-IWSP decisions.

4.4 Collisions

The allocation policy must consider scenarios where a query plan consists of multiple independent branches that may execute simultaneously, e.g., using the same Google example in 4.3 above, a query which spell-checks N tokens and retrieves the results of K searches. In this case, the independent branches of the query tree may compete for the same collection of nodes, resulting in collisions that degrade response time.

With IWSP, there are two forms of collisions: inter- and intra-Operation collisions (Note: According to WSDL standard Operations are defined as functions inside a single Web Service. Single Web Service may expose more than one operation). Inter-operation collisions occur when the allocation policy assigns two invocations referencing two different operations to the same node. These operations might be implemented by either (a) the same WS residing on a node (such as doSpellSuggestion and doGoogleSearch of Google) or (b) different WSs assigned to the same node (such as Google map [34] Web Service residing on the same node as the WS implementing the doGoogleSearch). Intra-Operation collisions occur when two invocations of the same operation reference the same node and compete for its resources. An example is two invocations of Google's doSpellSuggestion referencing the same replica of Google Web Service.

The ideal situation for execution is to have IWSP with enough replicas and smart allocation policy in the broker that collisions do not occur. However, in real life this is almost an impossible situation since deploying replicas is a costly operation. I will show in the following chapters that collisions do actually degrade the performance.

Chapter 5

Proteus: a query execution framework using Web Services

I mentioned in the chapter 1, that Web Services could be used a building blocks for Internet Database Management System (IDBMS). To facilitate this vision, one must have a complete framework that could handle the necessary steps starting from composing the plan, submitting it, executing it, and finally returning the results back to the client. In chapter 4, I investigated many design alternatives which include:

- Choice of allocation algorithm
- Broker Deployment
- Intra-Web Service Parallelism

In this chapter I will show a design of the Proteus Runtime Integration Framework[21], which is a framework for dynamic composition and execution of plans using Web Services[1]. I will use it to analyze the alternatives and derive conclusions with supporting experimental results.

Proteus consists of many components, namely:

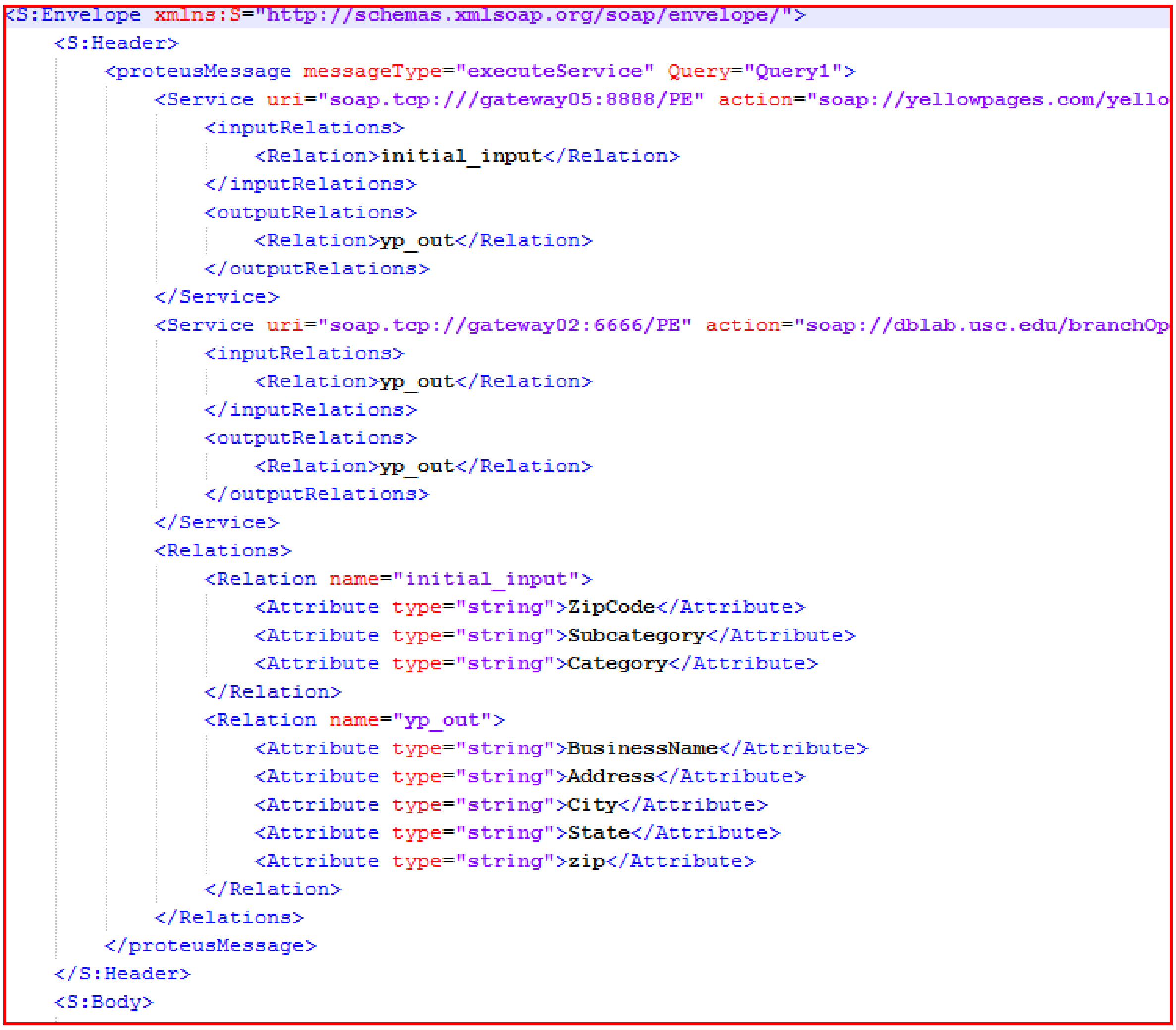
1. The mediator
2. The Coordinator
3. The Broker
4. A set of Proteus Operators

[1] Proteus was built in University of Southern California's Database Laboratory and I am a key member of the team who built it

In the remaining of this chapter I will explain each one of these components and explain how they interact with each other.

5.1 The mediator

A Framework that provides dynamic execution of plans must allow users or clients to issue their queries in many formats. Some of the suggested formats are SQL[SQL_REF], Web Service Flow Language(WSFL) [14], XL[13]. However, Proteus Framework (like any other framework) requires plan to be defined in a specific format called Proteus XML Plan. The mediator is the component of the system that receives the user/client query and translates it from its original language to the Proteus XML Plan. A sample Proteus XML plan should look like the following



```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header>
        <proteusMessage messageType="executeService" Query="Query1">
            <Service uri="soap.tcp:///gateway05:8888/PE" action="soap://yellowpages.com/yello
                <inputRelations>
                    <Relation>initial_input</Relation>
                </inputRelations>
                <outputRelations>
                    <Relation>yp_out</Relation>
                </outputRelations>
            </Service>
            <Service uri="soap.tcp://gateway02:6666/PE" action="soap://dblab.usc.edu/branchOp
                <inputRelations>
                    <Relation>yp_out</Relation>
                </inputRelations>
                <outputRelations>
                    <Relation>yp_out</Relation>
                </outputRelations>
            </Service>
            <Relations>
                <Relation name="initial_input">
                    <Attribute type="string">ZipCode</Attribute>
                    <Attribute type="string">Subcategory</Attribute>
                    <Attribute type="string">Category</Attribute>
                </Relation>
                <Relation name="yp_out">
                    <Attribute type="string">BusinessName</Attribute>
                    <Attribute type="string">Address</Attribute>
                    <Attribute type="string">City</Attribute>
                    <Attribute type="string">State</Attribute>
                    <Attribute type="string">zip</Attribute>
                </Relation>
            </Relations>
        </proteusMessage>
    </S:Header>
    <S:Body>
```

Listing 8: Sample Proteus XML Plan

As shown in Listing 8, the plan consist of a <ProteusMessage> element which is the main XML element. It holds a set of <Service> and <Relation> elements. <Service> elements specify the Web Services that are required for the execution of the plan, where <Relation> element specifies the format of the data exchanged between Web Services included in the plan. One can envision <Relation> element as the database structure of all the datasets used during the execution life cycle. Each <Service> contains <InputRelations> and <OutoputRelations> elements, showing the inputs and outputs of each Web Service. Both <InputRelations> and <OutputRelations> are mapped to a <Relation> element separately. This is used mainly to facilitate means of parameter matching to insure that the output of a certain Web Service can be plugged as an input in the next Web Service. It allows for count-matching (number of parameters) and type-matching (insuring that a parameter is of the same type as the Web Service expects).

A specific type of the <Service> element is the Proteus Operators themselves (I explain the Proteus Operators below). In that case the XML element must contain other sub-elements that control the flow of data and control the operation of the operator. For example, when SELECT operator is used, the select condition must be included in the XML Plan. When JOIN operator is used, the join condition must be indicated in the plan and so on.

5.2 The Coordinator

The coordinator represents the start and end points of the plan execution. The Proteus XML plan is passed to the coordinator as a SOAP envelope, where the header contains the plan and the body contains the relations. It consumes the XML plan submitted by the mediator, which is specified as a list of XML elements.

Each element specifies the Proteus Operator or a Web Service and its input and output relation(s). The coordinator executes the XML plan by traversing it. It recursively removes that top element of the XML plan, executes it, places its output in the body of the resulting SOAP envelope, and then executes the remaining elements. Proteus operators are implemented as linked-in libraries into the coordinator. Each is implemented as a function that accepts a SOAP envelope and produces one or more SOAP envelopes (see below).

During the execution and when the coordinator reaches an autonomous Web Service in the plan, it contacts the broker with the name of the Web Service (which is used as an ID for the Web Service) and the number of invocations to that Web Service(represented by the number of the records in the input relation). The broker replies with a list of URLs of replicas and the number of invocation for each.

If the coordinator does not have the proxy for a specific replica, it contacts the broker requesting its WSDL. Upon receiving the WSDL, the coordinator compiles it at runtime and uses it to invoke the Web Service. Dynamically generated proxies are stored for future references. The coordinator utilizes the Iterator operator to invoke Web Services. I explain how Iterator operator works in section 5.4.

The last step of execution is when all <Service> elements are completely consumed, and the Coordinator then returns the output of execution in XML format for the client that submitted the query in the first place.

5.3 The Broker

According to W3C standards for Web Services, each WS has a set of operations. The broker stores Performance Metadata (PM) at the operation level. For example, Google's Web Service has, among others, doGoogleSearch and doSpellSuggestion operations. Despite the fact that they are implemented in the same Web Service, the former operation is more resource consuming that the latter, since it requires searching Google's database that contains billions of pages. That is why it is unrealistic to allocate request to the two operations assuming they are identical, even they are hosted in the same Web Service.

The broker is a repository that contains the lists of Web Service replicas and their PM profiles. The broker component allows the service providers to register replicas of their Web Services. The service provider provides the broker with the WSDL URL of each Web Services replica. The broker downloads the WSDL and then asks the service provider for permission to build a performance profile for the replicas. The performance profiles are used by the broker to, intelligently, allocate requests to replicas. The broker allocates Web services intelligently when requested by the coordinator. The broker allocates replicas using the Least

Response Time (LRT) policy (in the following chapter I will show why LRT was chosen). To describe LRT, we must first provide a precise definition of response time (RT). We define response time as the elapsed time from when a client invokes the WS replicas till it obtains the results. This includes the network time, processing time, and possible queuing delays at the broker and WS replica. The current implementation deals with cases where replicas are hosted in the same network subnet which means the network part of the RT equation is identical for all replicas and hence is not considered. Future research eliminates the assumption that all replicas are within the same subnet and incorporate network delay in the RT equation.

The broker has two phases: registration phase and mid-flight phase.

5.3.1 Registration Phase

The registration phase occurs when the service provider submits their Web Services replicas' information with the broker. The information contains the Web Service name, its URL, and its WSDL (which contains the definition of the operations in the Web Service along with their inputs and outputs and their types). It is during this phase the broker asks permission to build the performance profile for each WS replica. This process is detailed below.

The broker utilizes a PM profile to facilitate intelligent allocation for Web services. The profiling process is directed toward examining the effect of concurrent invocations of Web Services on the throughput and response time measures for that Web Service. Throughput is defined as the number of requests processed by a Web Service replica per unit of time. We must distinguish between average and median response times and chose the median response time as an indication of the Web Service response time. Median response time give a more accurate indication of the average response time because it excludes extreme cases where the response time is either too high or too low. These cases can be caused by uncontrollable factors. For example delays observed in the operating system and different network layers. Operating system delays are the result of context switches between different threads of the RDBMS clients when concurrent invocations occur.

The profiling process in the broker simulates concurrent invocations by spawning a thread for each concurrent request. It repeatedly increases the number of concurrent invocations, i.e, threads, and observes how it impacts the throughput and the response time. Increasing the number of concurrent invocations results in increased throughput. However, this does not hold indefinitely. The profiler reaches a point where the throughput levels and increasing the number of concurrent invocations results in reduced throughput. This is attributed to the fact that context switching [25] overhead becomes dominant.

It is at the end of the profiling process where the Broker estimates the response time of a Web Service replica and uses it for allocation purposes. This response time is identified as T_{ST} in section 4.1.4

For certain applications, another mean of optimization could be accomplished by utilizing the Split operator (described below) to access partitioned data sources by routing SOAP envelopes to different replicas based on some selection criteria. This allows the service provider to partition their databases without reducing the ease-of-use of their Web Services.

5.3.2 Mid-Flight phase

In the mid-flight phase, the broker receives queries from the coordinator and provides WS allocation based on the performance profiles created at the registration phase.

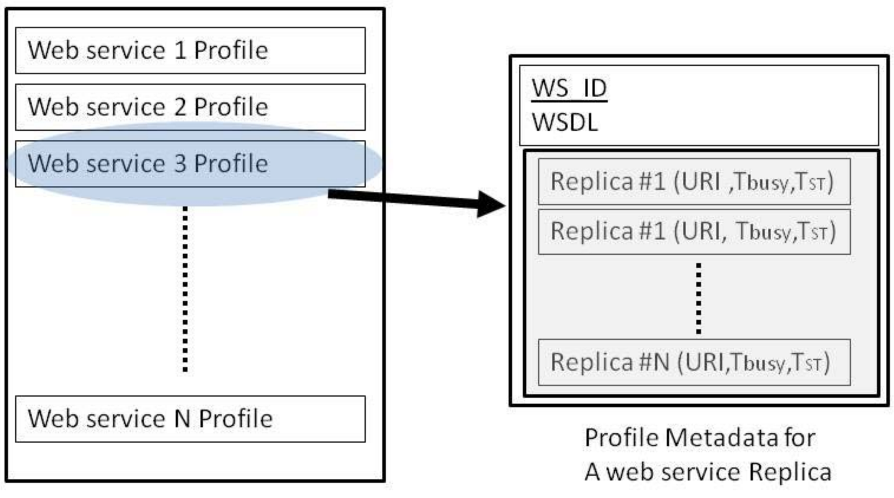


Figure 9: broker structure

5.4 Proteus Operators

To guarantee a complete execution of the plan, Proteus utilizes two classes of Operators (Implemented as Web Services):

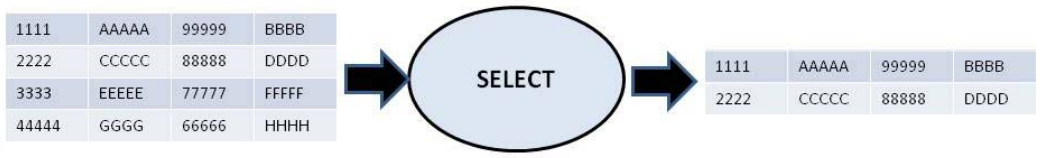
1. Standard relational algebra operators:
   1.1. SELECT,
   1.2. PROJECT,
   1.3. JOIN, and
   1.4. UNION
2. Operators that control flow of the data and control between Web Services included in the plan. These operator are:
   2.1. Branch,
   2.2. Split, and
   2.3. Iterator,

Each of these operators is explained in details below.

5.4.1 SELECT

The select operator is used to retrieve a set of tuples from a relation that satisfy the selection criteria. The selection criteria contains a comparison method based on the type of the data in the relation. For example, numerical data usually employ comparison like <, >, =, !=, while string data usually utilize comparison like substring and string inclusion. For example, assume querying a

Yellow Page Web Service and obtaining a list of businesses with the type of “Restaurant”. One could do a select operator for those with ZipCode =’90025’



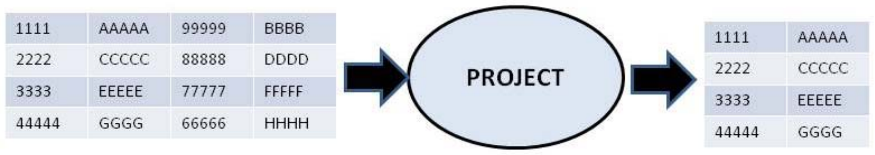
1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD
3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD

Figure 10: SELECT operator

5.4.2 PROJECT

The PROJECT operation is used to select a subset of the attributes of a relation by specifying the names of the required attributes. For example assuming a relation that contains information about a person, one might use the PROJECT operator to extract the first name attribute. This operator is used mainly to get around the parameter mismatch problems where the output of one Web Service contains more attributes than the input of the next one, in that case, the project operator is used to filter out unnecessary attributes.



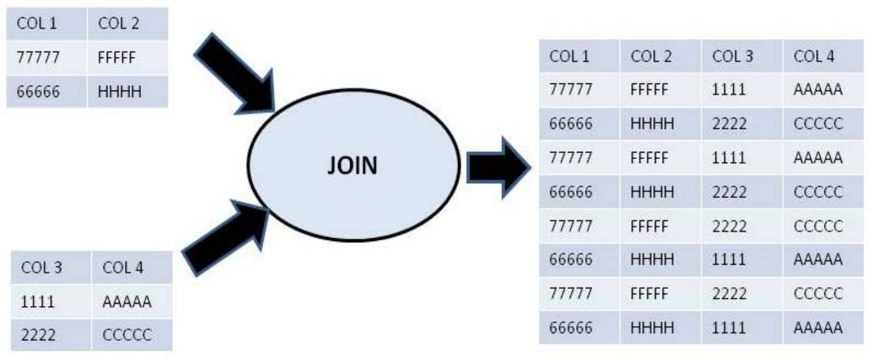
1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD
3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

1111	AAAAA
2222	CCCCC
3333	EEEEE
44444	GGGG

Figure 11: Project Operator

5.4.3 JOIN

JOIN is used to combine related records from two relations. While in its simplest form the JOIN operator is just the cross product of the two relations, it could become more complex resulting in records being removed within the cross product to make the result of the join more meaningful. Usually JOIN allows evaluating a join condition between the attributes of the relations on which the join is undertaken.



COL 1	COL 2
77777	FFFFF
66666	HHHH

COL 3	COL 4
1111	AAAAA
2222	CCCCC

COL 1	COL 2	COL 3	COL 4
77777	FFFFF	1111	AAAAA
66666	HHHH	2222	CCCCC
77777	FFFFF	1111	AAAAA
66666	HHHH	2222	CCCCC
77777	FFFFF	2222	CCCCC
66666	HHHH	1111	AAAAA
77777	FFFFF	2222	CCCCC
66666	HHHH	1111	AAAAA

Figure 12: JOIN operator

5.4.4 UNION

UNION operator allows records from two relations to be combined with the duplicate elimination. It is required that the two relations to contain the same number and type of attribute for the UNION operator to be executed. This operator is helpful in case of parallelism, where multiple requests are sent to different replicas, and after they are finished, they are combined together using the union operator.

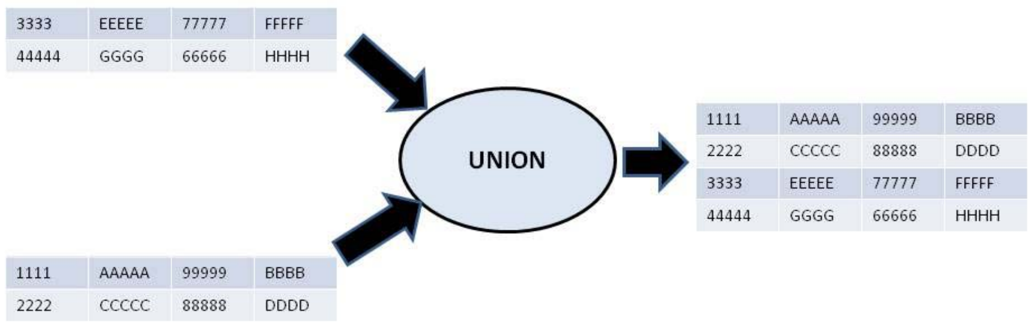
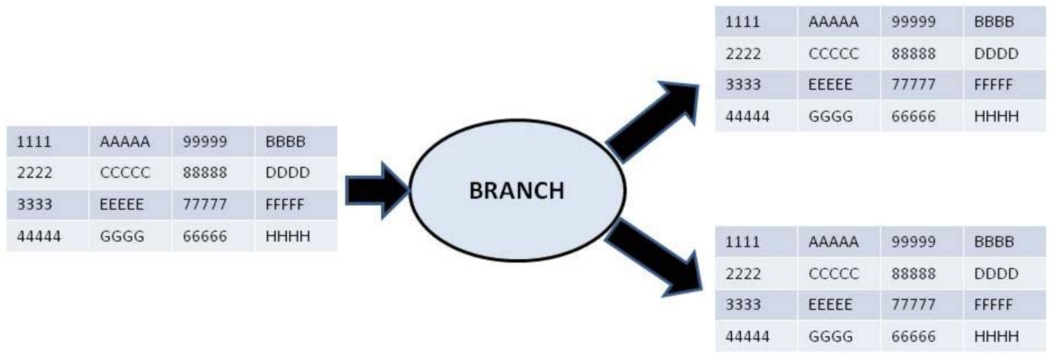


Figure 13: Union Operator

5.4.5 Branch

Branch operator allows the coordinator to send the same relation as input to two, or more, different Web Services at the same time. This is utilized when the plan contains two independent paths of execution, in which case the execution could be carried on parallel, allowing for better performance.



1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD
3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD
3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

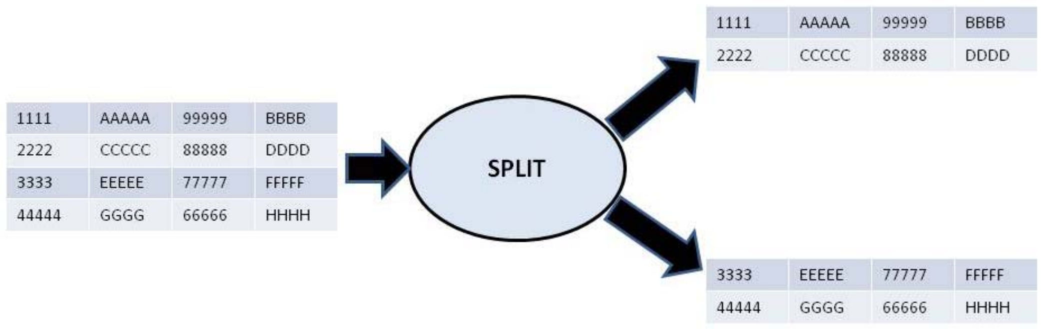
1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD
3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

Figure 14: BRANCH operator

Notice that the parallel paths must be combined at some part of the plan before the end of execution. Combining the branches is done using either the JOIN or UNION operators.

5.4.6 Split

Split operator works in a similar manner as the branch operator. The only difference is that the Branch operator sends the same relation to all branches, while the split operator employs a mechanism to separate the tuples (records) in the relation and send different records to different Web Services in parallel paths.



1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD
3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

1111	AAAAA	99999	BBBB
2222	CCCCC	88888	DDDD

3333	EEEEE	77777	FFFFF
44444	GGGG	66666	HHHH

Figure 15: SPLIT operator

5.4.7 Iterator

While Proteus operators accept a set of records (tuples), autonomous Web Services may only accept one input at a time. The Iterator operator is used to send a set of records to autonomous WSs one record at a time. This is because an autonomous Web Services may accept only one XML element (instead of a set) as input. Iterator invokes Web Services in two different ways: without Intra-Web Service parallelism (No-IWSP), and with Intra-Web Service Parallelism (IWSP). No-IWSP invokes Web Services one record at a time, one replica at a time, resulting in a completely sequential execution.

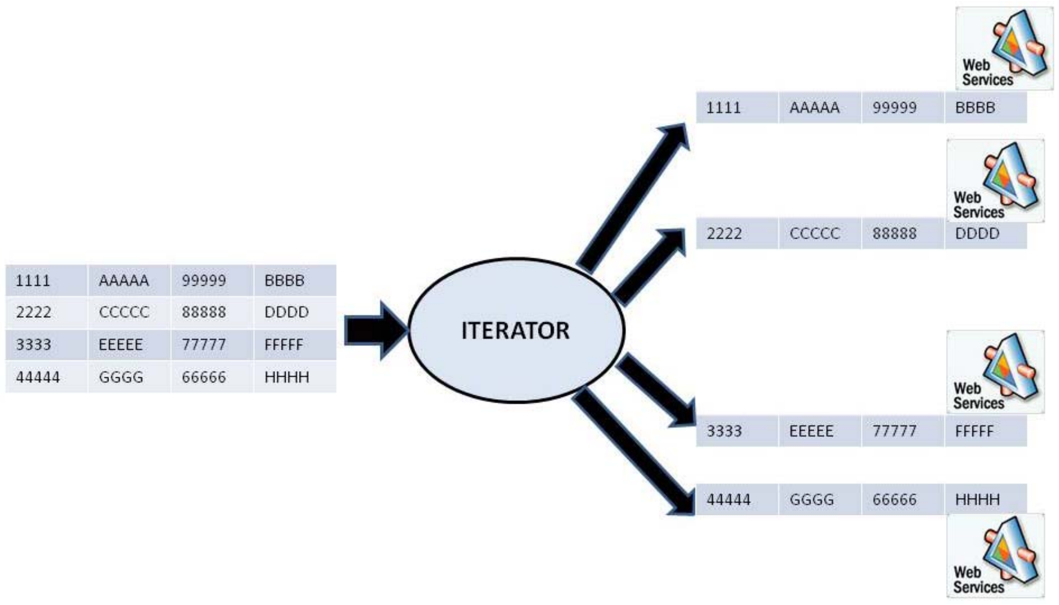
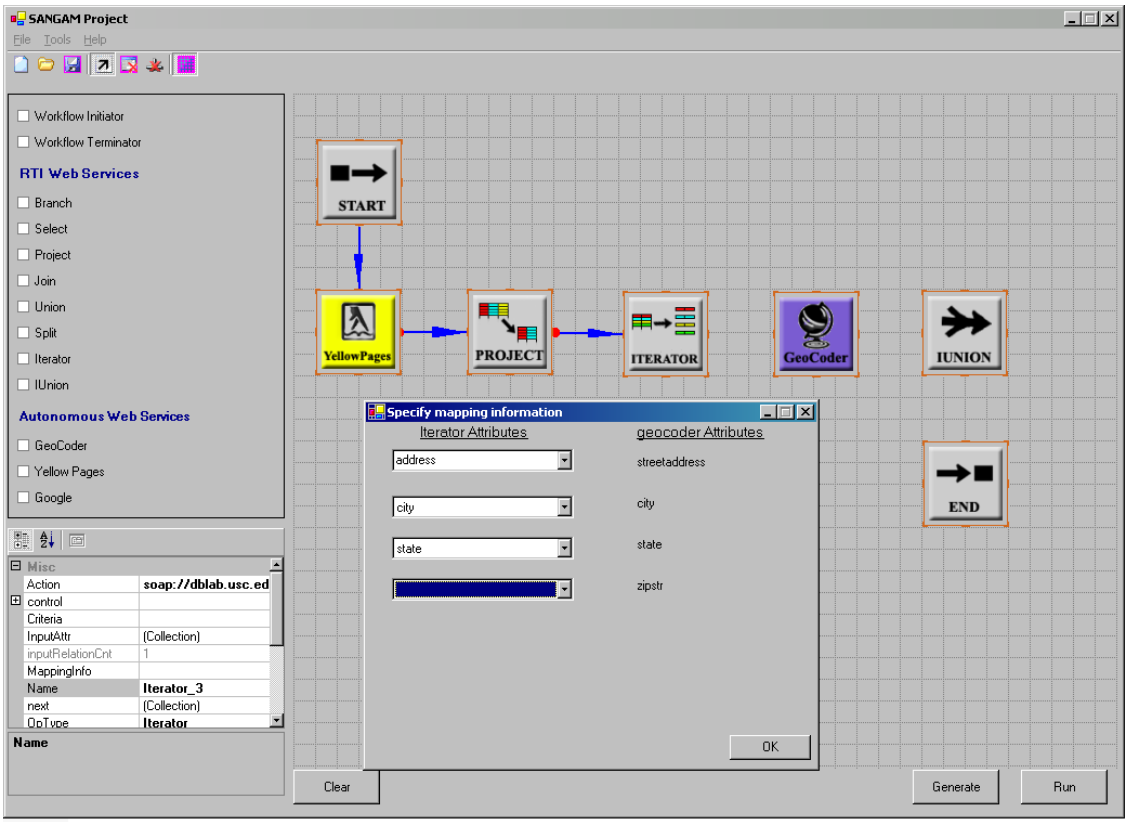


Figure 16: ITERATOR operator

IWSP, in the other hand, allows the execution engine to spawn a thread for each replica of the Web Service, and each thread invokes a Web Service replica with records sequentially. I will highlight the impact of IWSP and No-IWSP in the next chapter.

5.5 Plan Composition GUI

While it is not an essential component of the Proteus Framework, a Graphical User Interface (GUI) for Plan composition is considered to be a recommended addition to the framework. In this section, I will describe the Proteus GUI which allows the user to: a) easily define and add Web Services, b) compose a plan using an easy-to-use interface, c) submit the initial input, and d) finally submit the plan along with the inputs to the coordinator to carry on the execution and obtain the results.



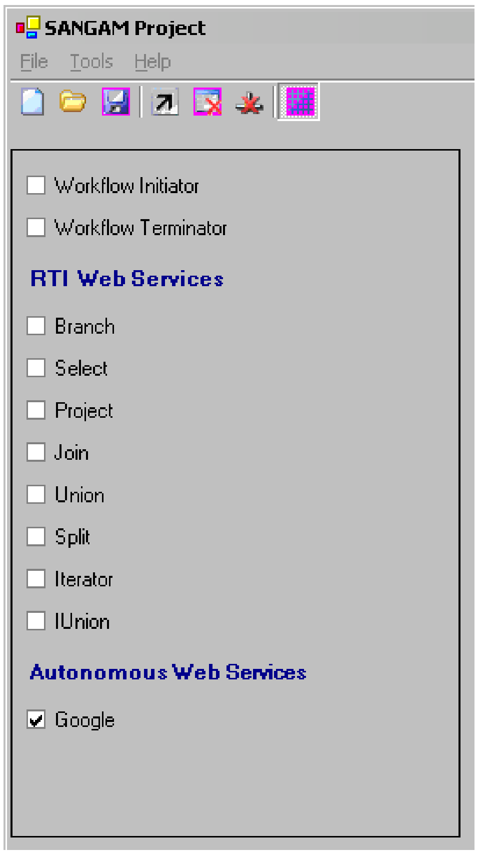
Iterator Attributes	geocoder Attributes
address	streetaddress
city	city
state	state
	zipstr

Misc	
Action	soap://dblab.usc.ed
control	
Criteria	
InputAttr	(Collection)
inputRelationCnt	1
MappingInfo	
Name	Iterator_3
next	(Collection)
OpType	Iterator

Name

Figure 17: Plan Composition GUI

The plan generator provides the user with the capability to integrate Web Services and to generate an XML plan based on the query specified. The plan generator provides a drag-n-drop interface to allow the user to generate Web Service integration plans. The user creates a workflow diagram, which is essentially a graphical query involving the autonomous Web Services and Proteus operators and invokes the coordinator to execute it.



- [ ] Workflow Initiator
- [ ] Workflow Terminator

RTI Web Services

- [ ] Branch
- [ ] Select
- [ ] Project
- [ ] Join
- [ ] Union
- [ ] Split
- [ ] Iterator
- [ ] IUnion

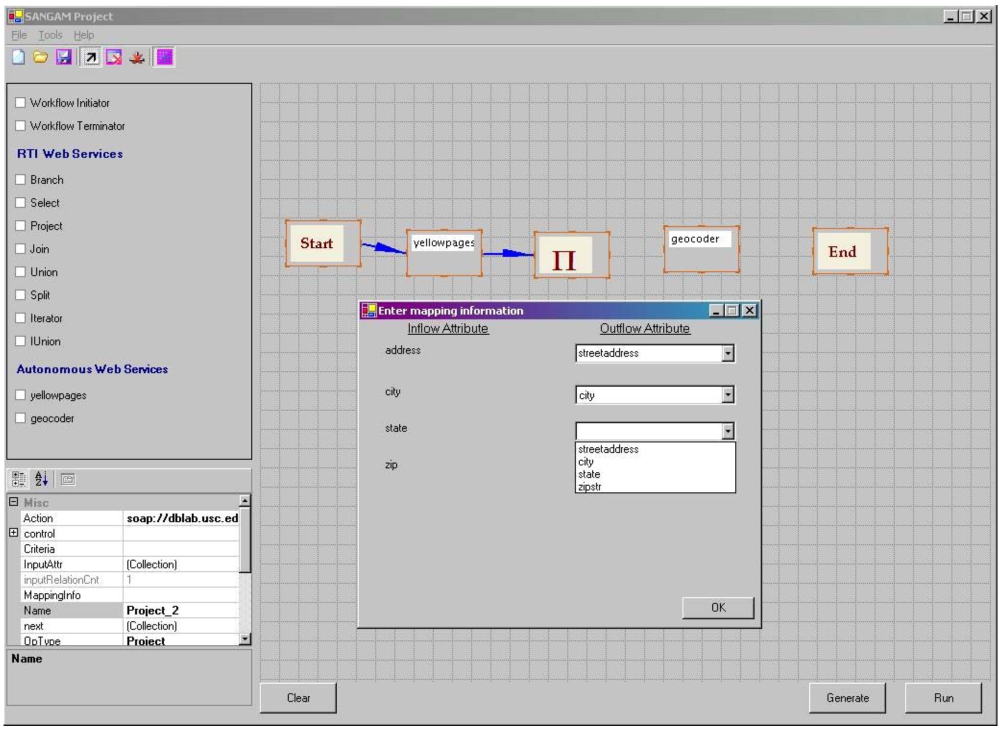
Autonomous Web Services

- [✓] Google

Figure 18: Selection of Operators or Web Services in Proteus

The user can register a new Web Service with the interface by specifying the URL of the Web Service. The tool extracts the operations specified in the WSDL and exposes them to the user. The user is required to select an operation for a Web Service (Figure 18).

The user can also save plans to their hard drive and recall them later for execution. In case the user has unregistered a Web Service from the interface, the software takes care to see that the execution of saved plans that reference the unregistered Web Services is not impacted.

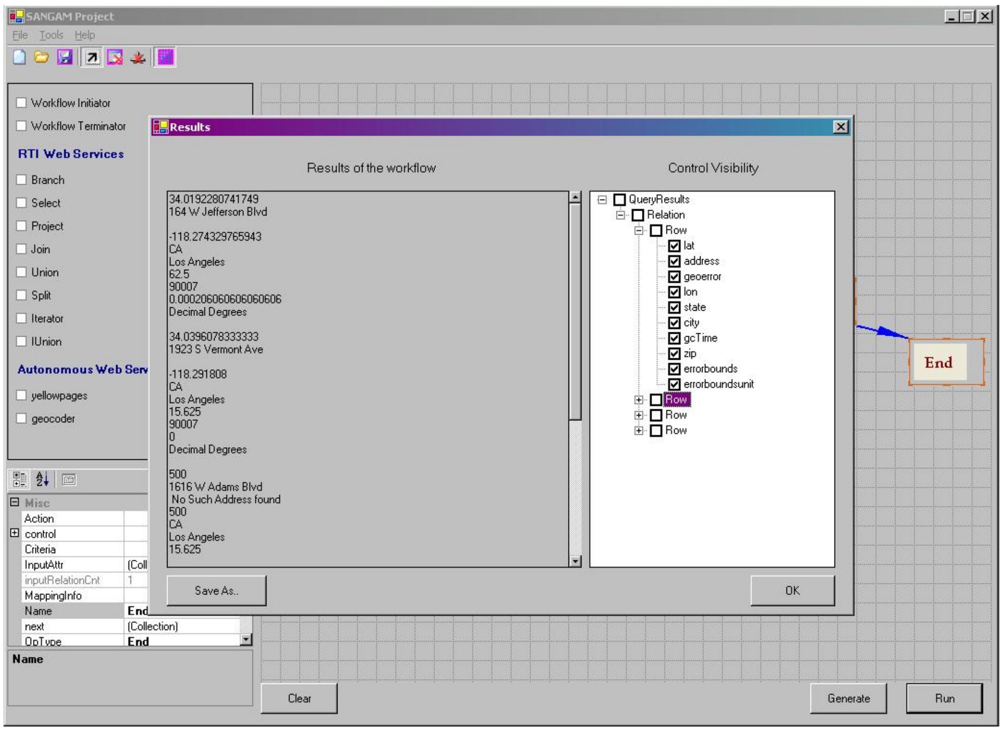


Inflow Attribute	Outflow Attribute
address	streetaddress
city	city
state	
zip	

Misc	
Action	soap://dblab.usc.ed
control	
Criteria	
InputAttr	(Collection)
inputRelationCnt	1
MappingInfo	
Name	Project_2
next	(Collection)
OpType	Project

Figure 19: Mapping metadata between Web Services

The tool allows for signature matching between Web Services. Signature mismatching happens when the number and/or the data type of the input and output flow for the Web Services in consideration does not match. PROJECT operator is used between the two Web Services to circumvent this problem. In order to solve the problem where the XML element names of one Web Service does not match with the XML element names of the next Web Service, the tool asks the user for the mapping information (see Figure 19)



```
34.0192280741749
164 W Jefferson Blvd

-118.274329765943
CA
Los Angeles
62.5
90007
0.000206060606060606
Decimal Degrees

34.0396078333333
1923 S Vermont Ave

-118.291808
CA
Los Angeles
15.625
90007
0
Decimal Degrees

500
1616 W Adams Blvd
 No Such Address found
500
CA
Los Angeles
15.625
```

Figure 20: Showing the results of Plan execution

Once the plan execution is completed, the GUI tool allows the user to view the result of the plan execution as a list (extracted from the resulting XML relation). This is shown in figure 20

Chapter 6

Analysis and Evaluation

In the previous chapters, we discussed the design of Proteus plan execution framework and provided many alternatives and decision and discussed them. In this chapter I will utilize this framework for experimental purposes and provide findings. Most of the issues investigated are presented as discussion in chapter 4.

First I will explain the experimental environment setup and then I will explain the assumptions made and, finally, provide the observations and lessons.

6.1 Experimental Environment

The framework and its components are developed using Microsoft .NET environment [23]. For the experiments, I will be using three data sources, namely:

1- Yahoo yellow pages (YP): This Web Service contains business information obtained from Yahoo!'s Yellow Pages service. The information consist mainly of the name of the business, its address (City, street address, zip code), and phone number.

2- GeoCoder (GC): This is a Web Service developed in Information Science Institute (ISI) and it performs the conversion of a street address to its corresponding GPS identifiers (Longitude and Latitude)
3- Tigerlines (TL): This is a Web Service that extends the information provided by GC by providing extra GPS information which is the Longitude and Latitude of the start and the end of the street.

Each of the three data sources is exposed as a Web Service with operations allowing querying of their database. It is important to notice that the nature of the Web Services and the information provided by each are irrelevant to the experiment. However, I chose them because a) I have access to them and can provide them in a controlled lab environment, and b) they present realistic, large data which makes the finding of the experiments apply to real-life applications.

I will also use three types of queries. Each query will help in investigating the aspects I discussed in chapter 4. The queries are described as follows:

1- Query 1 (Q1): This query extends information provided by YP with the longitude and latitude information of the business address, obtained from GC. The client uses this information to show the location of the business on an aerial map obtained from any Map service like Google Maps[34] or Microsoft Bing Maps[35].

2- Query 2 (Q2): This query extends Q1 by employing TL to add the street information (start and end longitude and latitude) to the result.

3- Query 3 (Q3): This query is the same as the Q2 with one difference. The TigerLine database is fragmented so that each US state information is deployed in a standalone TL Web Service. This means the plan must query each TL individually and union their output.

Each query plan employs Proteus specific WSs that serve as the glue between the autonomous WSs. These are categorized into two groups. The first corresponds to standard relational algebra operators: select, project, join, etc. The second implements data-flow-and-control operators such as Branch and Split (Please see Chapter 5 for more detailed description of Proteus and its operators). These resemble simpler versions of the Exchange operator and Eddies [15,26]. The Branch WS constructs k copies of its XML formatted input data and forwards a copy to a different pre-specified WS. The Split WS is provided with a selection clause and a destination Web Services for the qualifying XML elements. It consumes its XML formatted input, applies the selection clause, and forwards all the qualifying elements to the specified target WS. Figure 21 shows three different query plans.

The first employs a geo-coder to show the location of the business on the map. It employs the Branch operator in combination with Project because the GC Web Service accepts only the street address, city, state and zip of the business, i.e. phone number and other information cannot pass through. The second plan repeats the same with TL because it accepts only the same address information as GC. The third query employs the split operator to send each address to the TL Web Service in the corresponding state. The results of all states are combined together using the Union Web Service which performs duplicate elimination.

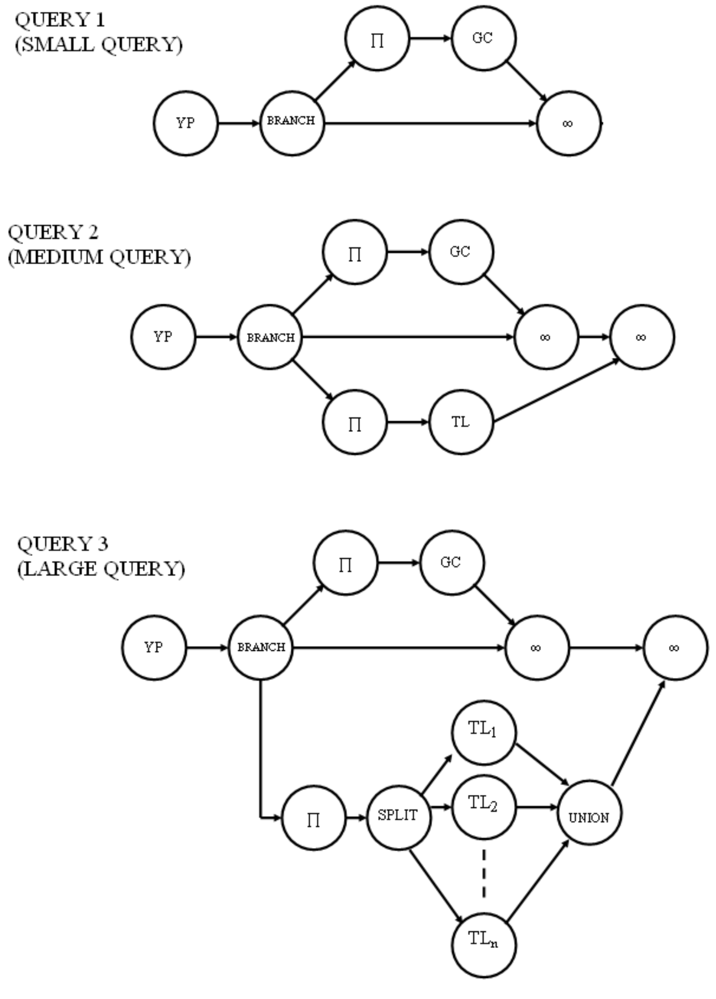


Figure 21: Three queries used to evaluate the four policies: Small (Query1), medium (Query2) and large (Query3) queries

6.2 Experiment preparation

In order to reach the observations I present in this chapter, I did the experiment in two phases. The first phase was a simulation phase and the second phase was an experiment in a controlled lab environment. The two phases complement each other, and the results from the first one is used, as well as validated, in the second.

6.2.1 Phase 1: Simulation

Examining the allocation policies requires experiment with larger number of servers (up to 40) to host Web Service replicas and flexibility to change the parameters of experiments. Due to the difficulty on achieving this level of control, the decision is made to use a simulated environment.

To obtain an accurate and realistic estimation of service times, the Proteus implementation is used to execute the three queries in a small lab environment where only one copy of a WS is present. For each experiment, more than 1000 queries are executed and service times for the invoked Web Services are recorded (The mean service time is chosen). The queries were executed for machines with different CPU/Memory configuration to allow capturing the aspects of homogeneous and heterogeneous environments. both homogeneous and heterogeneous environments were both considered. The homogeneous environment consists of machines with 2.0 GHz processors. The heterogeneous environment consists of a uniform mix of 5 different processor speeds: 1.4, 1.6, 2.0, 2.4, 3.0 GHz. The service times observed from these experiments are plugged into the simulator and used to compare the different combinations of allocation and scheduling policies. We Assume fully connected network topology with fixed delay between any two nodes.

There are two forms of collisions in our environment. The first, termed intra-WS collisions, occurs when the same copy of a Web Service (say GeoCoder) is utilized simultaneously, resulting in formation of queues. The second, termed inter-WS collision, refers to the scenario where requests reference different WSs (e.g., GeoCoder and Yahoo Web Services) hosted on the same node (resulting on requests competing for the machine resources). A collision is further categorized as either inter-plan or intra-plan. With inter-plan collisions, different plans compete for the same node. With intra-plan collisions, branches of the same plan might compete for the same node at the same time. All four possible collision types are captured by the simulator.

An open simulation model is employed, where requests arrive at a pre-specified rate (λ) using a Poisson distribution. Where request probability is define as :

$$P(n) = \frac{(\,)^{n}}{n!}\, e^{-(}$$

Since Proteus is a distributed system, a naïve policy that results in many plans colliding on the same node (due to inter and intra-WS/plan collisions) will cause that node to become a bottleneck at a certain arrival rate. This node becomes fully utilized with many queued up requests while other nodes sit idle waiting for work. At this point, the observed execution times are dependent on the implementation of the employed random number generator and are un-reproducible. This simulation state is termed undesirable. A technique that supports the highest arrival rate prior to the simulator becoming undesirable is superior.

A plan might be scheduled either statically or dynamically. With a static allocation, the Web Services referenced in a plan are allocated when the plan is submitted for execution and before the execution starts(i.e. once the plan stars execution, no reference to the broker is made). This makes plan execution simpler, since intermediate Web Services do not participate in node allocation. However, this reduces the efficiency of Web Service allocation criteria because the information used to make the decision might be outdated after the execution of the plan starts (which is observed easily in higher arrival rates).

With a dynamic allocation, the Web Services are allocated on demand by conceptualizing a producer/consumer relationship between Web Services. A consuming Web Service is allocated before a producing Web Services finishes execution. This requires intermediate nodes to participate in selecting the copy of next Web Services (for example, in Q1, after YP Web Service produces its results it queries the broker for the replica of GC Web Service to send its results to). Requiring intermediate Web Services to participate in the allocation process is an overhead, but the allocation decisions are more accurate and up-to-date, since the information in the lookup directory might have been updated after the plan execution started.

6.2.2 Phase 2: Controlled Lab experiments

In this phase, Proteus framework and autonomous Web Services (namely, YP, GC, and TL) are deployed in a controlled lab environment where machines are connected over Gigabit network switches and the queries are executed to observe the effects of different setups on a real-life environment where an off-the-shelf Operating System and DBMS software are used. This helps in examining and validating the observations made in Phase 1.

6.3 Discussion and observations

Note: References made from Figure 22 to Figure 29 are observations made from Phase 1. The rest are observation made from Phase2.

Figures 22 to 25 below show LRU is inferior to all other policies because it becomes unstable at a lower arrival rate. LRT is superior to all other polices when service time is estimated accurately (which is done in the preparation phase, see 6.2.1). When compared with dynamic scheduling, LRT's performance is inferior with static scheduling. This is because static scheduling invokes allocation of all Web Services that constitute a plan before the plan execution starts. This means that even though a Web Service is assigned to the plan at time t_1, it is not actually used till time t_2 ($t_2>t_1$). This increases inter-plan (both inter- and intra-WS) collisions, making the incurred service time at time t_2 higher than that estimated at t_1 due to queuing delays.

I was pleasantly surprised to find LRT performing well as long as the estimated response time of a Web Service is randomly distributed along a mean that matches the true service time of the Web Service. Figure 26 shows LRT's performance with different scheduling policies (static and dynamic) for different random distributions. Given a service time S, a random distribution of 70% corresponds to service times randomly picked from the range

S± 0.7S. Note that LRT's performance is degraded significantly if the estimated response times are completely random.

It is also observed that LU performs worse than LRA with static scheduling. This is because LRA does load balancing in a similar manner using the timestamps while LU uses the utilizations measurement which fluctuates frequently (leading to the frequent appearance and disappearance of temporary bottlenecks). Compared to LU, LRA does not require any network communication with broker because it requires no node characteristics to render a decision. This makes LRA more desirable than LU.

I also investigated the scalability of these policies and noticed that introducing more nodes in the system does not always improve performance. Figures 27 to 29 show the performance improvement of LRU, LRT and Random for three different configurations consisting of ten, twenty and forty nodes, respectively. Similar to prior experiments, Web Services are replicated across all nodes. LRU performance does not scale because it results in formation of bottlenecks with a low arrival rate. While Random is more scalable than LRU, a configuration deployed with LRT exhibits the best scalability characteristics. Similar trends are observed with both (1) homogeneous configurations and (2) static scheduling strategy.

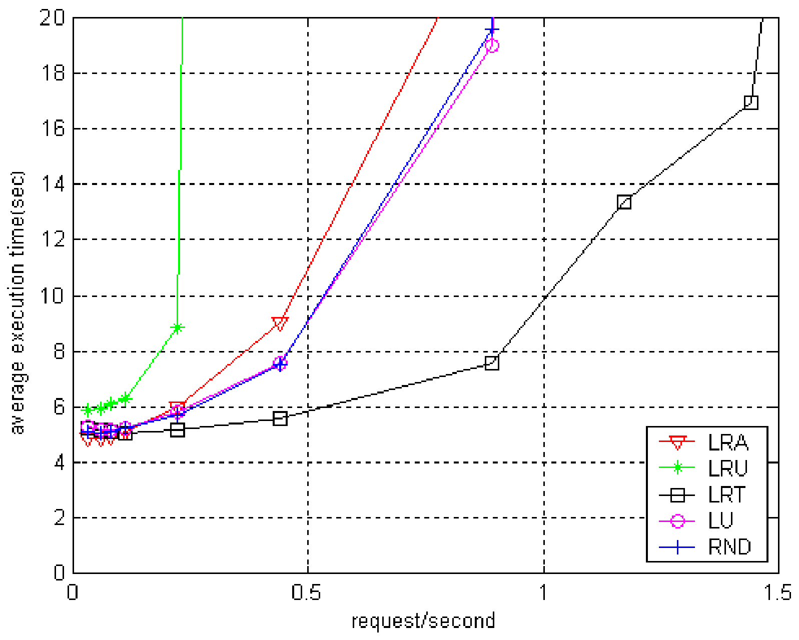


Figure 22: Average execution time of Q3 with alternative allocation policies, heterogeneous configuration using dynamic scheduling.

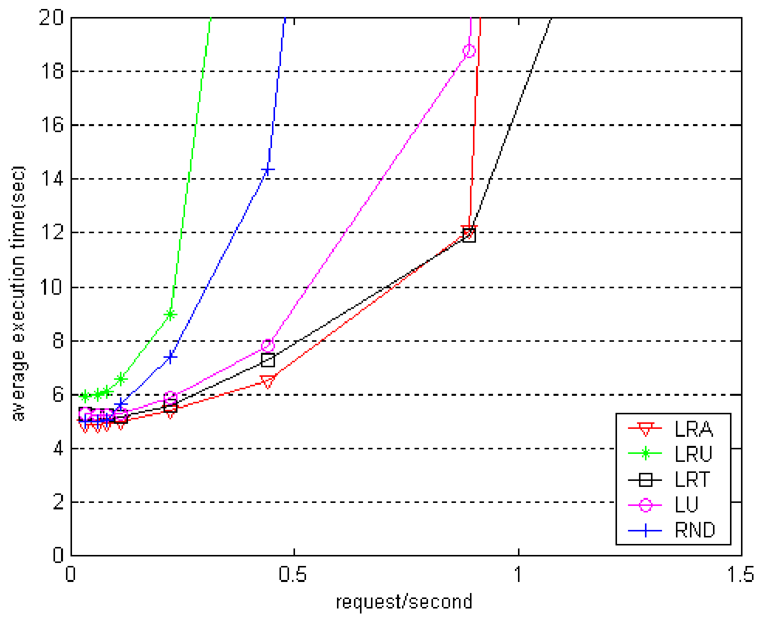


Figure 23: Average execution time of Q3 with alternative allocation policies, heterogeneous configuration using static scheduling.

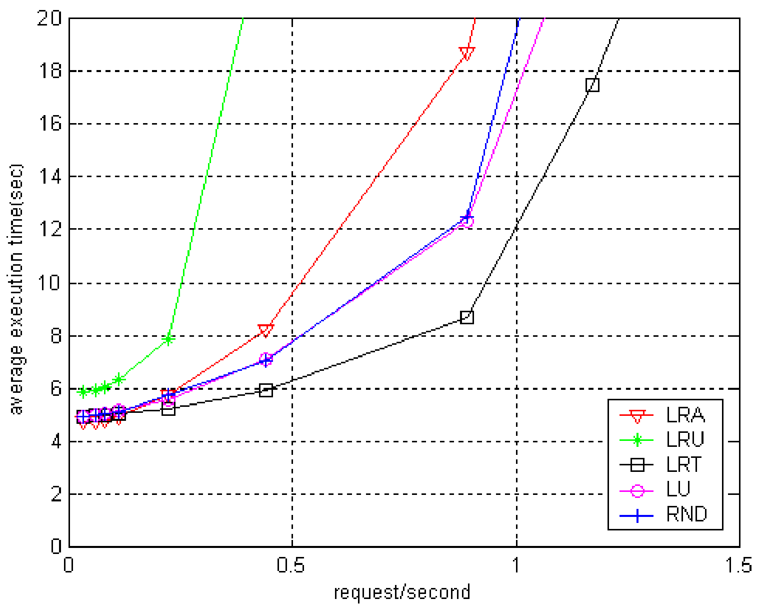


Figure 24: Average execution time of Q3 with alternative allocation policies, homogeneous configuration using dynamic scheduling.

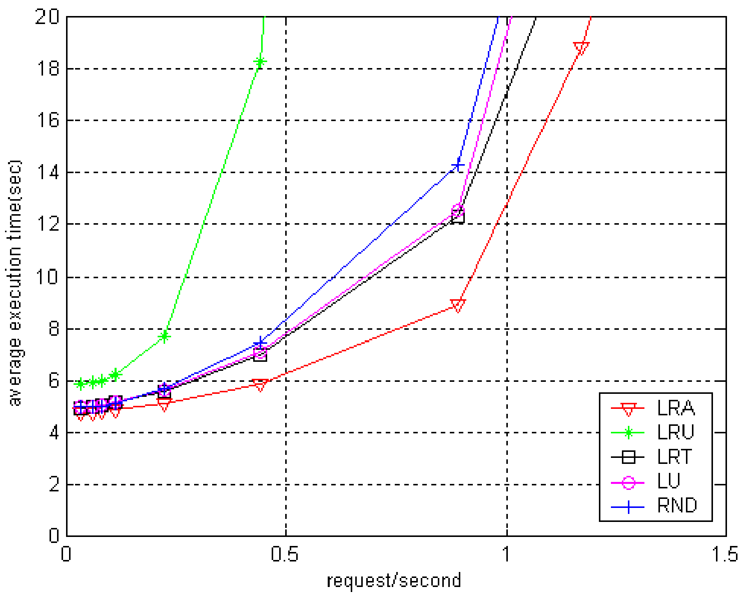


Figure 25: Average execution time for Q3 with a homogeneous configuration using static scheduling.

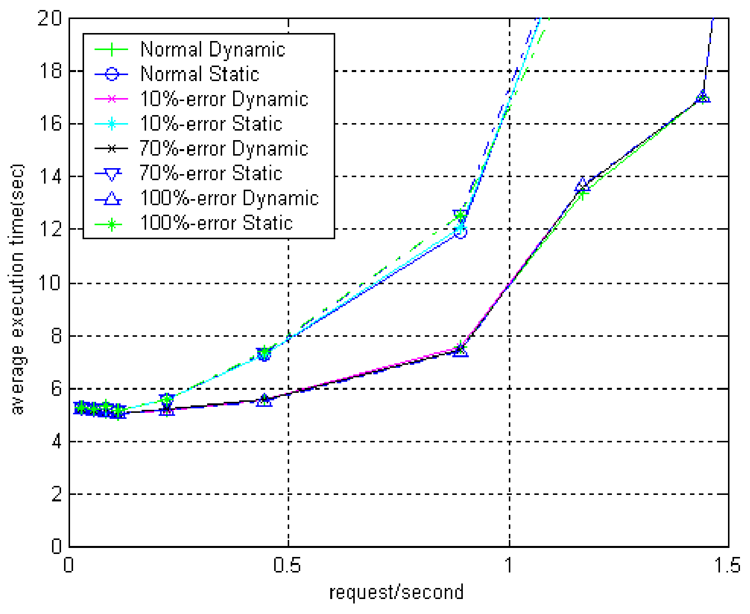


Figure 26:Average execution time for Q3 with LRT, heterogeneous configuration using different random distributions

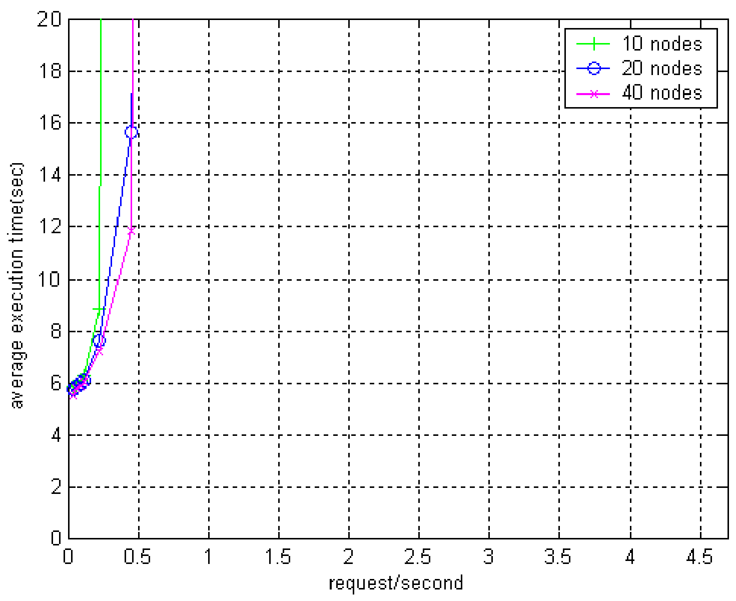


Figure 27: Scalability of a heterogeneous configuration with LRU allocation policy and dynamic scheduling

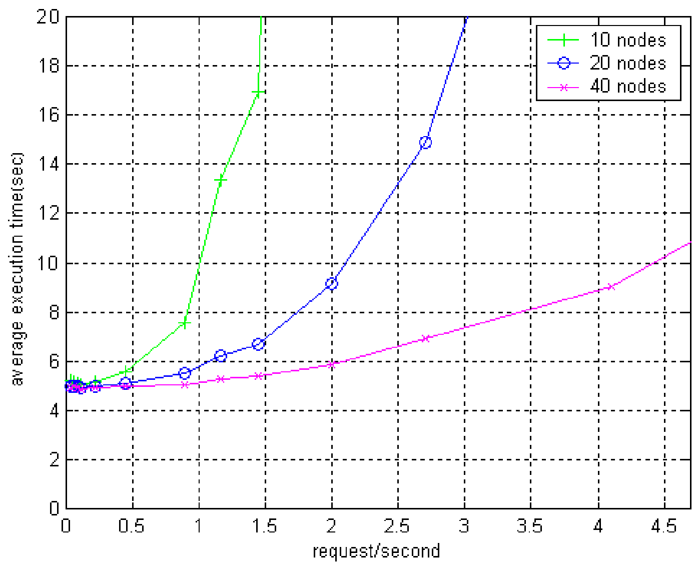
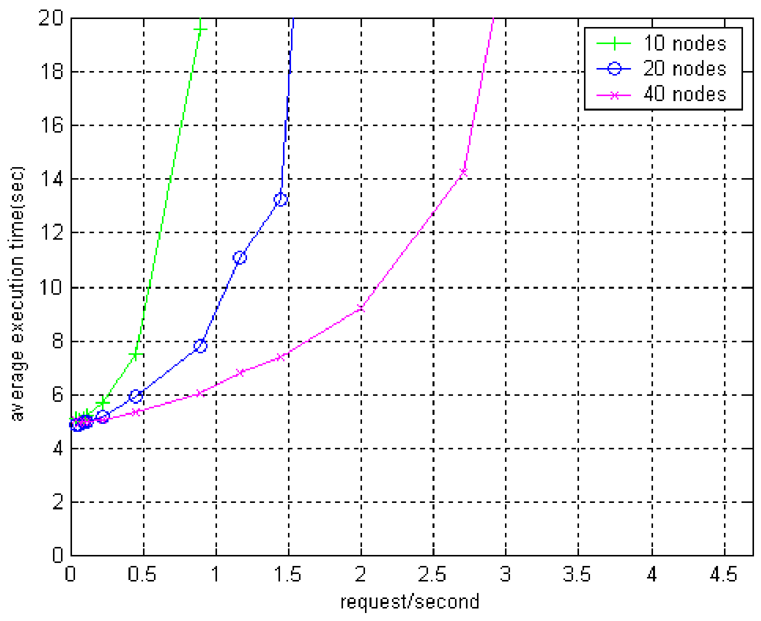


Figure 28: Scalability of a heterogeneous configuration with LRT allocation policy and dynamic scheduling



.

Figure 29: Scalability of a heterogeneous configuration with Random allocation policy and dynamic scheduling.

hese graphs above explain the choice made to use LRT as the allocation policy for the Broker component of Proteus Framework.

Further investigation is done to see the effect of collisions on the response time. I ran experiments using different placement of Web Service replicas across the PCs. Each Web Service provides only one operation. Hence, intra-operation collisions do not occur because the Iterator operation implemented by the centralized coordinator of Proteus does not issue a request for a Web Service until it has received results for its previously issued request. Moreover, the branches of a query (such as Query Type 3) do not include paths consisting of multiple sequential invocations that reference the same Web Service (for example, GeoCoder is not invoked by different branches of a query tree).

I analyzed a variety of WS assignments across the nodes. One assignment is to place copies of YP, GC, and TL WSs on mutually exclusive nodes. This would eliminate the possibility of inter-operation collisions. This configuration is analyzed in the context of **Observation 1** below (Figures 30 and 31). Another extreme is to assign a copy of each WS on each node, e.g. with one replica, they system consists of one node hosting all three WSs. This results in inter-operation collisions and constitutes the focus of all three observations presented below. There are other hybrid placements that were eliminated from this thesis because they did not provide additional observations beyond the three presented below.

Note: Observations and lessons discussed in this section apply to results obtained from experiments with all three query types. However, to avoid redundancy I present results from Query3. In other words, the observations presented below hold for all 3 query types.

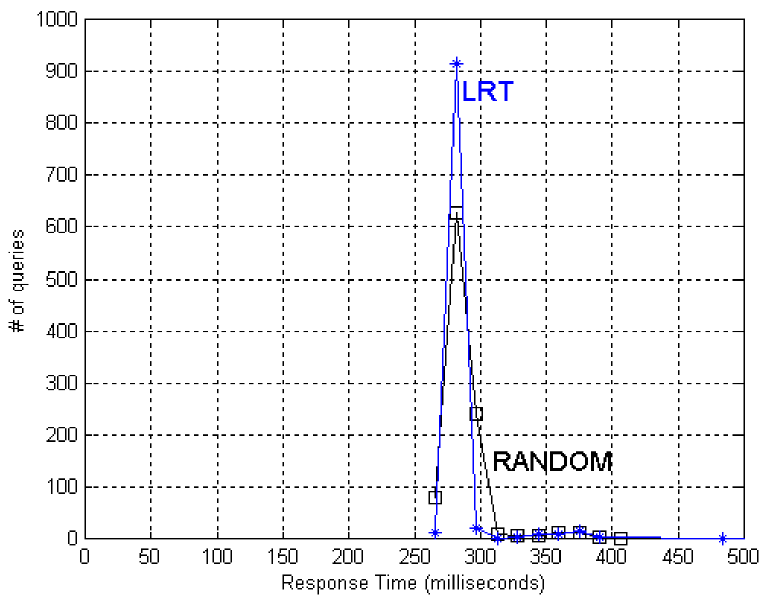


Figure 30: Distribution of RT of the query with 4 nodes and No-IWSP using ideal setup

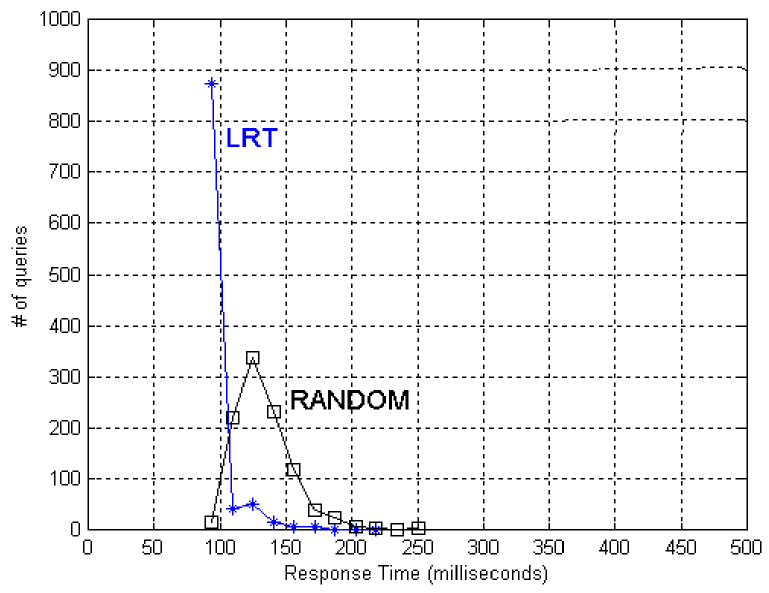


Figure 31: Distribution of RT of the query with 4 nodes and IWSP using ideal setup

Note: I ran the experiments in a homogeneous lab environment. I used a zero-load environment where there is only one query in the system at any given time and issued 1000 queries for each deployment.

The performance of different policies is compared by observing the median and distribution of the Response Time (RT) for 1000 queries submitted one after another. Figures 32 and 33 show the distribution of RT for the two policies deployed in a 4 replica configuration. The X axis shows the observed response time for Query type 3 and the Y axis shows the number of queries observing that specific response time. The ideal distribution is single point graph with zero-sized tails as its distribution. Such a graph is desirable because it produces a completely predictable system and insures that the response time observed is the actual, accurate, one.

This cannot be accomplished in real systems due to uncontrollable delays observed in the operating system and different networking layers. Operating system delays are the result of context switches between different threads of MySQL clients when Inter-Operation collision occurs. Also, when using IWSP, the coordinator encounters context switching delays between Iterator threads accessing different replicas. Network delays are attributed to delays caused by connection establishment and lost or erroneous packets.

To make this section readable, I will present obtained results in the context of three key observations learnt from our experiments. We focus on Query type 3 because the same observations are shown with different combinations of query types and deployments. This query retrieves a zip code that causes the YP Web Services to produce 16 objects. The observations are as follows.

6.3.1 Observation 1

LRT is superior to Random when the service time of Web Services is either known in advance or can be estimated with a high accuracy. By superior, I mean that its median response time is better and the behavior of the system is more predictable because a larger number of queries observe this median response time. This is particularly true when using IWSP. Figures 32 and

33 show this lesson with Query type 3. In these experiments, we use a cluster of 4 nodes and assign a replica of each WS on each node. Even with both IWSP and no-IWSP, LRT outperforms Random because it minimizes the number of inter-operation collisions. The impact of this collision is most evident with IWSP, see Figure 33. Here, Random results in a system that provides a response time varying from 140 to 200 milliseconds for most queries. With LRT, more than 80% of queries observe a response time of 125 milliseconds. The explanation for this is as follows. With Random, the centralized coordinator assigns a different number of objects to each Web Service replicas. While these are processed one at a time avoiding the overhead of multi-threading, the number of records assigned to each WS replica is not even, resulting in an unpredictable distribution of work across the WS replicas. LRT distributes the object requests across WS replicas based on their service time profiles, approximating an even distribution of WS invocations.

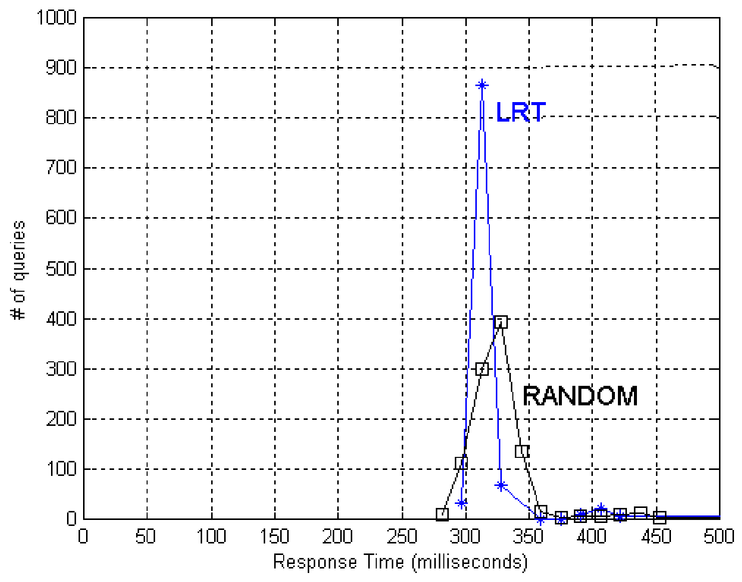


Figure 32: Distribution of RT of the query with 4 nodes and No-IWSP

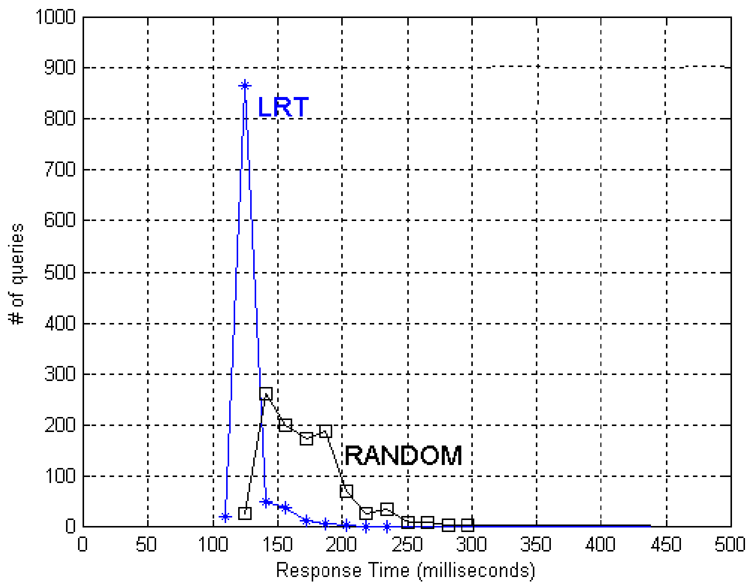


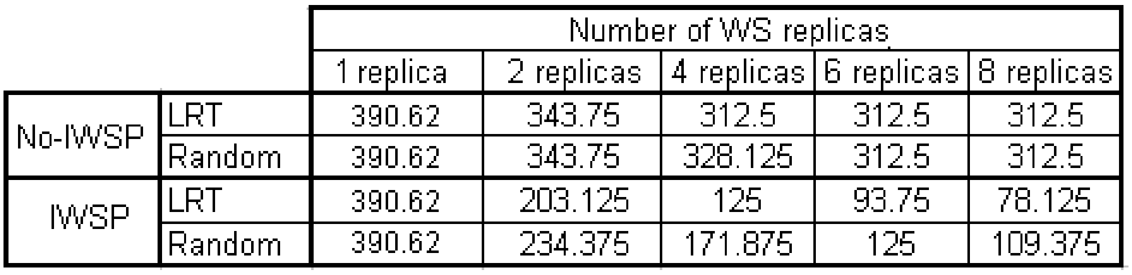
Figure 33: Distribution of RT of the query with 4 nodes and IWSP

To validate the claim that inter-operation collisions are the cause of discrepancy between LRT and Random, we constructed 4 replicas of each WS (YP, GC, and TL) on mutually exclusive set of nodes. This experiment uses all the twelve machines in the cluster and avoids inter-operation collisions. Figures 30 and 31 show the obtained results with both no-IWSP and IWSP. When compared with the previous configuration, there is an improvement in the median response time (compare Figures 31 and 33). For example, with LRT and IWSP, the median response time of 125 milliseconds is reduced to 95 milliseconds, an improvement of 30%. Random observes the most improvement with its response time becoming more predictable. Note that LRT continues to outperform Random because it does a better job of distributing the 16 invocations evenly across the nodes.

6.3.2 Observation 2

With no-IWSP, multiple replicas of a WS do not enhance the response time significantly and, hence, the choice of an allocation policy is not important. Note that this lesson is in the context of response time for a zero-load system. It will most likely prove to be false when considering system throughput.

Listing 9 shows this lesson by reporting the median response time with No-IWSP and IWSP with the alternative allocation policies. With No-IWSP, the response time is almost the same with both LRT and Random. Moreover, there is only a negligible improvement (less than 10%) when we increase the number of WS replicas from 2 to eight. This improvement is attributed to a lower number of inter-operation collisions due to a larger number of WS replicas.



		Number of WS replicas				
		1 replica	2 replicas	4 replicas	6 replicas	8 replicas
No-IWSP	LRT	390.62	343.75	312.5	312.5	312.5
	Random	390.62	343.75	328.125	312.5	312.5
IWSP	LRT	390.62	203.125	125	93.75	78.125
	Random	390.62	234.375	171.875	125	109.375

Listing 9: means for the RT values

6.3.3 Observation 3

With IWSP, while both Random and LRT allocation policies benefit from a larger number of WS replicas, one should not expect a linear increase due to serial processing times independent of the number of WS replicas.

Listing 9 shows that additional replicas have a dramatic improvement on response time with IWSP. With LRT, there is a factor of 2.6 improvement in response time as we vary the number of WS replicas from two to eight. It is a factor of 2.15 with Random. Figures 34 and 35 show the distribution of response time with alternative number of replications. Obtained results show LRT provides a more predictable response time.

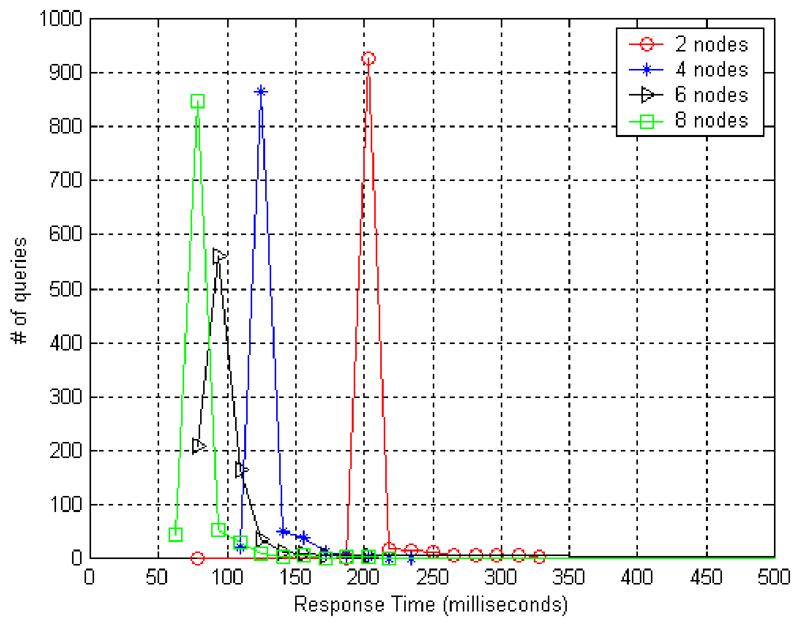


Figure 34: RT Distribution of query 3 with LRT and IWSP

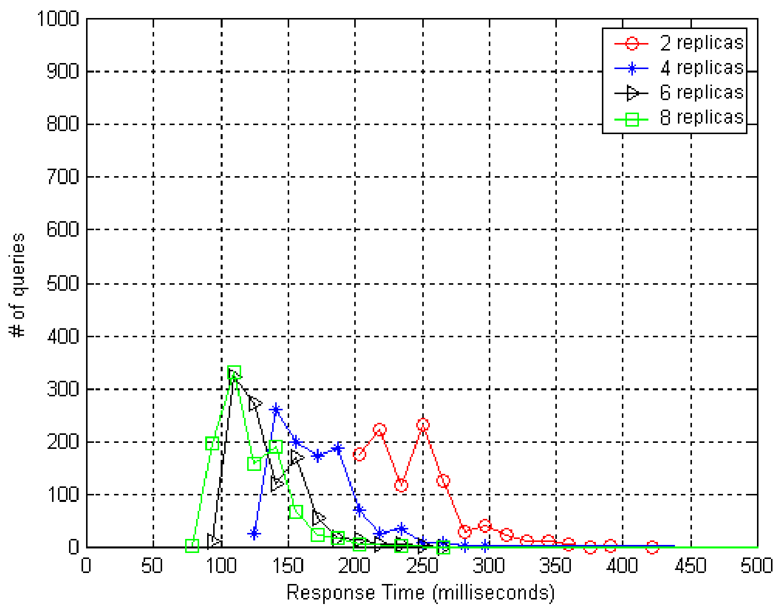


Figure 35: RT Distribution of query 3 with Random and IWSP

Both LRT and Random observe an improvement in their median response time because a larger number of Web Service replicas results in fewer inter-operation collisions. At the same time, one should not expect a linear (factor of 4 from two nodes to eight nodes) improvement in response time with a policy. This is because the number of Web Service replicas impacts the processing time of Geo Coder and Tiger Line WSs in Figure 21. The remaining operations are implemented in the centralized coordinator with a fixed processing time. Amdhal's law states that the speedup observed as a function of the number processors (replicas) is:

$$S = N \div [\ (B \times N) + (1 - B)\ ]$$

where B is the percentage of algorithm performed serially and N is the number of processors. If B is zero, S becomes N, resulting in a linear speedup. In our environment, B is greater than zero due to the serial processing of the centralized coordinator and invocation of YP Web Service. With one replica, we measure B to be 5%. Amdhal's law estimates a factor of 5.9 improvement with eight replicas. In practice, we observe a factor of 5 improvement (compare 390.62 to 78.125). this is attributed to inter-operation collisions increasing the value of B.

Chapter 7

Conclusion

This thesis discussed a component (named the “Broker”) that is used to allocate Web Service replicas in a framework for plan execution to insure better response time. To examine such component, the thesis outlined the design of the framework, described its components and showed some alternatives and decisions that should be considered with such framework.

I could summarize the contribution of the thesis by providing its main insights:

1. dynamic scheduling is superior to static. Results showed that dynamic scheduling provides better response time. This is attributed to the fact that dynamic scheduling uses more updated in formation in the broker which increases its accuracy.
2. LRT allocation policy is superior to the other examined alternatives. LRT incorporate more information in the Performance Metadata for web services allowing for better estimation of its utilization and, hence, better allocation decision.

3. A Web Service allocation policy must consider the utilization of nodes when multiple copies of different Web Services are assigned to the same node. In the experiments, LRU and LRA do not capture this important detail, providing inferior performance when compared to other policies (including a Random allocation policy).
4. Intra Web Service parallelism (IWSP) was presented as the means to enhance response time of a single query using multiple replicas of the Web Services in the query plan. The gain in response time does not scale exponentially as IWSP could result in more collisions when the request arrival rate is high.
5. LRT strives to minimize two forms of collisions (intra and inter operation) observed with IWSP in order to both enhance the median response time and the percentage of queries that observe this median.

Several assumptions were made to achieve the results in this thesis. Those assumptions could be eliminated by doing further research. Some of the future research topics that were not investigated in details in this thesis are:

1. Including the Network delay in the response time equation. All results provided in this thesis are done in a controlled lab environment where network delay is negligible.

2. Introducing more queries in the system and observing how it affects the performance for LRT and explain those observations.
3. A mechanism that enables accurate service time estimation for Web Services while taking background load into account.
4. Applying improvements on current implementation of the framework in general and the broker specifically. Examples are:
    a. Making the broker more resilient by incorporating solutions like Dew[10] to enable continuous operation in the presence of exceptions.
    b. Replicating the Broker's repository using solutions like Content Addressable Network (CAN) [9]. This guarantees data recovery in case of hard disk crash in some nodes.

The focus of this thesis was to discuss the allocation mechanism for Web Services to achieve better response time. All insights and observations we made by implementing the chosen allocation mechanism in the Broker component in Proteus plan execution framework and then investigate it using different experimental setups. However, The allocation mechanism could be plugged into any similar framework that has the allocation problem. The current implementation could be easily modified to define the broker component as a stand-alone component with a set of APIs defining how it interfaces with any framework.

Furthermore, one could envision the Broker component as a stand-alone service that is used by several frameworks (including Proteus) Simultaneously.

Proteus framework could be utilize to provide a “What”-oriented framework instead of a “How”-oriented. In other words, the framework could be extended with an additional component that takes the user query describing “what” he/she wants to accomplish and convert that query into a formulated XML plan and submitted to Proteus. This isolates all the technical details of how the query is formulated and executed from the user.

References

[1] Fan Li. Xu Zhang. Fan Zhang. and Yong Shan. *Grid-Based Digital Forestry Platform Dynamic Allocation of Resources to Web Services Management Research.* In ICCMS '10 Proceedings of the 2010 Second International Conference on Computer Modeling and Simulation - Volume 03.2010.

[2] Keyvan Mohebbi, Suhaimi Ibrahim, Mojtaba Khezrian, Kanmani Munusamy and Sayed Gholam Hassan Tabatabaei. *A comparative evaluation of semantic web service discovery approaches*. iiWAS '10 Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services. 2010..

[3] Yash Patel and John Darlington. *Average-BasedWorkload Allocation Strategy for QoS-Constrained Jobs In A Web Service-Oriented Grid*. In EDOCW '06 Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops. 2006

[4] Stephen S. Yau and Ho G. An. *Adaptive resource allocation for service-based systems*. Internetware '09 Proceedings of the First Asia-Pacific Symposium on Internetware. 2009

[5] Yuxiong He, Sameh Elnikety, and Hongyang Sun. *Tians Scheduling: Using Partial Processing in Best-Effort Applications*. ICDCS '11 Proceedings of the 2011 31st International Conference on Distributed Computing Systems. 2011

[6] Marvin Ferber, Sascha Hunold and Thomas Rauber. *Load Balancing Concurrent BPEL Processes by Dynamic Selection of Web Service Endpoints*. ICPPW '09 Proceedings of the 2009 International Conference on Parallel Processing Workshops. 2009

[7] E. Alwagait and S. Ghandeharizadeh, *A Comparison of Alternative Web Service Allocation and Scheduling Policies*. IEEE SCC04, 319:326.

[8] Vladimir Stantchev. *Effects of Replication on Web Service Performance in WebSphere*. IBM Technical Report TR-08-003. March, 2008

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A Scalable Content Addressable Network*. In ACM SIGCOMM, 2001.

[10] E. Alwagait and S. Ghandeharizadeh. *DeW: A Dependable Web Services Framework.* In 14th International Workshop on Research Issues on Data Engineering, Boston, MA, March 2004.

[11] OASIS WebService Interactive Applications http://www.oasis-open.org/

[12] WS-Addressing specifications at W3C http://www.w3.org/Submission/ws-addressing/

[13] D. Kossmann D. Florescu, A. Grnhagen. *XL: A Platform for Web Services*. In Conference on Innovative Data Systems Research (CIDR), January 2003

[14]Web Service Flow Language http://www.ebpml.org/wsfl.htm

[15] R. Avnur and J. Hellerstein. *Eddies: Continuously Adaptive Query Processing*. In Proceedings of ACM SIGMOD 2000.

[16] F. Banaei-Kashani, C. Chen, and C. Shahabi. *WSPDS: Web Services Peer-to-Peer Discovery Serivce*. In the International Symposium on Web Services and Applications (ISWS'04), June 2004.

[17] S. Decker, S. Melnik, F. Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann and I. Horrocks. *The Semantic Web: The Roles of XML and RDF*. IEEE Internet computing, 15(3), 2000.

[18] http://www.webopedia.com/TERM/W/Web_services.html

[19] http://www.gartner.com

[20] World Wide Web Consortium http://www.w3.org/2002/ws/

[21] S. Ghandeharizadeh, C. Knoblock, C. Papadopoulos, C. Shahabi, E. Alwagait, J. Ambite, M. Cai, C. Chen, P. Pol, R. Schmidt, S. Song, S. Thakkar, and R. Zhou. *Proteus: A System for Dynamically Composing and Intelligently Executing Web Services*.

In the First International Conference on Web Services (ICWS), Las Vegas, Nevada, June 2003.

[22] J. Klein F. Leymann S. Thatte F. Curbera, Y. Goland and S.Weerawarana. *Business Process Execution Language for Web Services, Version 1.0.* July 2002.

[23] Microsoft Corporation. Microsoft .NET Framework.

[24] Microsoft Corporation. Global XML Web Services Architecture (GXA), 2002.

[25] B. Albahari, P. Drayton, and B. Merrill. C# Essentials.

O'Reilly, 2001.

[26] F. Tian and D. DeWitt. *Tuple Routing Strategies for Distributed Eddies*. In Proceedings of VLDB 2003, Berlin,Germany, September 2003.

[27] D. Box. *Understanding GXA*. Microsoft Corporation. URL: http://msdn.microsoft.com/webservices/understanding/gxa/default.aspx. July 2002.

[28] Kelvin K. Yue and David J. Lilja. *Dynamic Processor Allocation with the Solaris Operating System*. Parallel Processing Symposium. March 1998.

[29] Regiane Y. Kawasaki, Luiz Affonso Guedes, Diego L. Cardoso1, Carlos R. L. Francˆes, Glaucio H. S. Carvalho, Jo˜ao C. W. A. Costa and Nandamundi L. Vijaykumar. *A Markovian*

Performance Model for Resource Allocation Scheduling on GNU/Linux. ISPA Workshops pp844-853. 2006

[30] S. Ghandeharizadeh, C Papadopoulos, P. Pol, and R. Zhou. *NAM: A Network Adaptable Middleware to Enhance Response Time of Web Services*. In International Journal of Web Services Research (JSWR), Volume 2, Issue 4, October 2005

[31] UDDI Community. *UDDI Version 2.0 Data Structure Specification.* UDDI Organization, June 2001.

[32] UDDI Community. *UDDI Version 2.0 Programmer's API Specification*. UDDI Organization, June 2001.

[33] SANGAM, http://dblab.usc.edu/Sangam

[34] https://maps.google.com

[35] http://www.bing.com/maps/

[36]Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[37]Simple Object Access Protocol (SOAP), http://www.w3.org/TR/soap12-part1

[38] Flow-based Infrastructure for Composing Autonomous Services (FICAS). http://eil.stanford.edu/ficas\