A HYBRID ARCHITECTURE TO IMPROVE SOAP PERFORMANCE IN WEB SERVICES COMMUNICATION

MUTANGE KENNEDY SENAGI

MASTER OF SCIENCE

(Software Engineering)

JOMO KENYATTA UNIVERSITY OF

AGRICULTURE AND TECHNOLOGY

A hybrid architecture to improve SOAP performance in web services communication

Mutange Kennedy Senagi

A thesis submitted in partial fulfillment for the degree of Master of Science in Software Engineering in the Jomo Kenyatta University of Agriculture and Technology

2014

DECLARATION

This thesis is my original work and has not been presented for a degree in any other university.

……………..……………………… ……………………………………….

Signature Date

This thesis has been submitted for examination with my approval as university supervisor.

……………..……………………… ……………………………………….

Signature Date

Dr. George Okeyo

This thesis has been submitted for examination with my approval as university supervisor.

……………..……………………… ……………………………………….

Signature Date

Dr. Wilson Cheruiyot

DEDICATION

I dedicate my thesis work to my family and many friends. A special feeling of gratitude goes to my loving parents Daniel and Leah Senagi whose words of encouragement, guidance, prayers and wisdom have enabled me to come this far. Special dedications also go to my brothers, sisters and friends who have always supported me throughout the process. Thank you all! God bless you!

ACKNOWLEDGEMENTS

I would like to express my gratitude and appreciation to all those who supported me to complete this thesis. Special thanks to my supervisors Dr. Okeyo and Dr. Cheruiyot for their wisdom, exemplary guidance and mentoring. I would also like to thank Dr. Kimwele, the Associate Chairman of Department (ACOD) JKUAT Westlands Campus, whose help, motivation, suggestions and encouragement played a tactical role for me to make out this research. Special thanks also go to Sati Arthur and Philip Chunju for their constructive brainstorming sessions, discussions, and documentation review. Finally, I would also want to appreciate Dedan Kimathi University of Technology, Department of Information Technology, for providing me with resources that aided me to realize this research's experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ACRONYMS

SOA	Service oriented architecture
WS	Web service
SOAP	Simple object access protocol
REST	Representational state transfer
QoS	Quality of service
WWW	World Wide Web
W3C	World Wide Web Consortium
XML	Extended markup language
HTTP	Hypertext transfer protocol
HTTPS	Hypertext transfer protocol secure
UDP	User datagram protocol
SMTP	Simple mail transfer protocol
CORBA	Common object request broker architecture
RMI	Remote method invocation
COM	Component object model
UDDI	Universal description discovery and integration
WSDL	Web service description language
RDF	Resource description framework
DAML	DARPA agent markup language
RPC	Remote procedure call
TCP	Transport control protocol
EDI	Electronic data interchange

DUT	Data update tracking
MTOM	Message transmission Optimization mechanism
IIS	Internet information services
IDE	Integrated development environment
OS	Operating system
API	Application Programming Interface

ABSTRACT

A web service is an implementation of service-oriented architecture which extends World Wide Web infrastructure thus providing means of integrating software applications in loosely coupled distributed systems. It provides an abstract interface in which applications access services via ubiquitous web protocols and data formats such as HTTP, XML and SOAP. Web services communication is aided by Simple Object Access Protocol (SOAP). SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed computing environment using XML. SOAP beats its competitors like CORBA and Java RMI due to its simplicity, flexibility, robustness, extensibility and inter-operability in heterogeneous systems. However, SOAP transmits its data represented in XML. XML documents are huge in size and verbose thus becoming a major hindrance in performance for high performance applications that process lots of data. This research improves the performance of XML-based messaging technique in SOAP communication model of applications implemented in web services. To improve SOAP performance, a combination of techniques was integrated which include: client-side caching, simple server-side database queries, compression technique and document-literal style description of WSDL. A relatively high turn-around time and lower network throughput is recorded. Nevertheless, performance of SOAP is improved in terms of bandwidth utilization and transfer time while running SOAP web service applications. This can be useful in disadvantaged networks (10mbps) and subsequently save costs in communication.

CHAPTER ONE

INTRODUCTION

1.1 BACKGROUND INFORMATION

The internet is growing very fast, it has become an important tool in communication, providing services and sharing information. Many organizations, institutions and individuals have embraced internet in many ways e.g. e-commerce, blogs etc. Such services can be provided in the internet by using Service-Oriented Architecture (SOA), Web Oriented Architecture (WOA) etc. SOA is being adopted by many programmers as a way of integrating software systems and providing different services thus building dynamic systems that are loosely coupled (Bianco et al., 2007). A loosely coupled system is one which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

There are various technologies that implement SOA, including Common Object Request Broker Architecture (CORBA) (IONA Technologies, 2005), Java Remote Method Invocation (RMI) (Grosso, 2001), Component Object Model (COM) (Microsoft, 2013) and web services (Booth et al., 2004). There are various advantages and disadvantages of adopting either of these technologies in implementing SOA. Web service implementation using SOAP gives it the capability of being platform independent and the ability of going through firewalls without being recognized. SOAP messages are encapsulated within HTTP. HTTP is a universal standard in the World Wide Web. Firewalls by default allow traffic through port 80 which HTTP uses in communication. Moreover, web services work in heterogeneous systems; this makes it stand out amongst its equals which are monolithic in nature (Coulouris et al., 2009).

A web service provides a network accessible interface of functionality/methods for an application. It is built using standard internet technologies e.g. e.g. HTTP, XML, SMTP, Jabber etc. From a programmer's perspective, a web service is an abstract layer invoked to provide services; separating programmers from web service specific details (Snell et al., 2001). A web service is invoked programmatically to provide services to the software developer as shown in Figure 1.1.

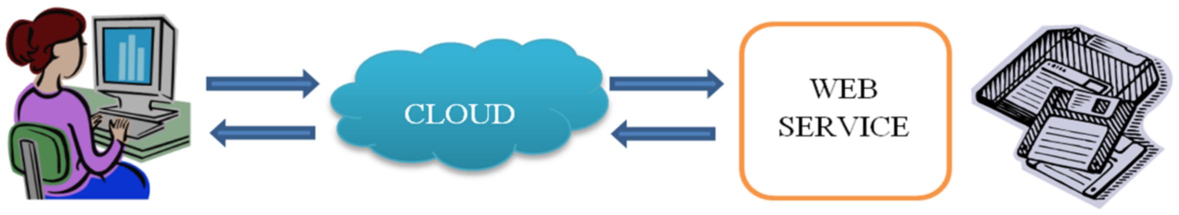


Figure 1.1: Web services abstract layer in the cloud

(Adopted from Snell et al. (2001))

A web service interface generally consists of a collection of interfaces that can be used by a client over the internet. Operations in a web service may be provided by different resources, for example, programs, objects or databases. A web service may either be managed by a web server along with other pages; or it may be a totally separate service (Coulouris et al., 2009). Web services are platform independent and can be invoked from any programming language. Client applications, like web browsers, that understand these technologies (HTTP and HTML) can interact with the web service efficiently (Snell et al., 2001) (Coulouris et al., 2009).

The key characteristic of web services is that, they can process XML-formatted SOAP messages. Each web service uses its own service description to deal with the service-specific characteristics of the message it receives (Coulouris et al., 2009). Communication in web services is enhanced by SOAP that does the packaging of the actual message being transmitted. SOAP relies on XML in formatting the messages.

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC (Don et al., 2000).

SOAP uses XML to represent messages and HTTP in communication in web services. Programmers do not normally need to be concerned with these details. This is as a results of SOAP Application Programming Interfaces (APIs) which have been implemented in many programming languages, including Java, JavaScript, Perl, Python, .NET, C, C++, C# and Visual Basic. To support client-server communication, SOAP specifies how to use the HTTP POST method for the request message and response for the reply message. The combination of XML and HTTP provides a standard method for client-server communication over the internet (Coulouris et al., 2009).

Both SOAP and the data it carries are represented in XML – a textual self-describing format. Textual representation takes up more space than binary ones and requires more space to process. In document style interactions, speed is not an issue, but is important in request-reply interactions. However, it can be argued that there is an advantage in human-readable format that allows for easy construction of simple messages and for debugging more complex messages. It also allows a user to see the text in a message before it. But there are situations in which it is too slow (Coulouris et al., 2009).

Each item in an XML description is annotated with its type and the meaning of each type is defined by a schema reference within the description. This makes the format extensible, enabling any type of data to be transported. There is no limit on the potential richness and complexity of document format in XML, but there could be a problem for those that become unduly complex (Coulouris et al., 2009).

SOAP is a robust and extensible protocol for the exchange of messages. SOAP is the most widely used communication protocol in the web services model. SOAP's XML-based message format hinders its performance, thus making it unsuitable in high-performance scientific applications. The deserialization of SOAP messages, which includes processing of XML data and conversion of strings to in-memory data types, is the major performance bottleneck in a SOAP message exchange (Nayef et al., 2005).

1.2 PROBLEM STATEMENT

Web services extends World Wide Web infrastructure to provide means for software to connect to other software applications in loosely coupled distributed systems. Applications access web services via ubiquitous (universal) web protocols and data formats such as HTTP, XML and SOAP, with no need to worry about how web services are implemented (Microsoft, 2013). Communication in web services is aided by SOAP. SOAP is a standardized de facto XML-based protocol for packaging, services invocation and exchanging messages in web service interfaces. SOAP specifications defines nothing more than a simple XML-based envelope for information being transferred, and a set of rules for translating application and platform-specific data types into XML representations (Snell et al., 2001).

XML's structure is verbose (redundant textual characteristics and it uses tags to delimit the data) which generates huge documents. This results to considerable network traffic, poor bandwidth utilization and higher latency than competing technologies, like Java RMI and CORBA. Moreover, generation and parsing of SOAP messages and their conversion to-and-from in-memory application data is computationally very expensive in high performance applications resulting to high costs of implementing SOAP web-based applications. Therefore, SOAP's dependency on XML in communication reduces its performance in high performance applications (Coulouris et al., 2009) (Tekli et al., 2011) (Seyyed et al., 2011).

1.3 JUSTIFICATION

Organizations focus on making profit and reducing expenses. The cost of accessing internet is relatively high and end users usually want quick response to services they request while using software applications. SOAP is a technology used in implementing communication in web services. SOAP is standardized to do packaging of its messages using XML. However, SOAP-XML in communication is slow in high performance applications and its verbose structure consumes more bandwidth, which reduces the performance of SOAP (Coulouris et al., 2009).

This research improves performance of SOAP-XML messages by reducing the size of the messages being transmitted. This improves bandwidth utilization and reduces cost of running web services-based applications.

1.4 OBJECTIVES

1.4.1 GENERAL OBJECTIVE

This research investigates SOAP performance to improve its communication in web services. This is aimed at improving performance of SOAP communication channel in high performance applications in disadvantaged networks. This eventually improves bandwidth utilization and transfer time of SOAP messages in web services, hence improves the costs of running software applications.

1.4.2 SPECIFIC OBJECTIVES

(i) To investigate the relationship between SOAP and web services

(ii) To investigate SOAP performance bottlenecks

(iii) To identify techniques used to overcome SOAP performance bottlenecks

(iv) To develop a hybrid architecture that improves SOAP performance in communication

(v) To implement, test and evaluate the developed hybrid architecture

1.5 RESEARCH QUESTIONS

(i) What is the relationship between SOAP and web services?

(ii) What are some of the bottlenecks in SOAP performance?

(iii) What are some of the techniques that have been used to overcome SOAP performance bottlenecks?

(iv) Can we develop a hybrid architecture that improves SOAP performance in communication?

(v) How can we implement, test and evaluate the developed hybrid architecture?

1.6 SCOPE

This research investigates web services and its implementation in SOAP. We then give an insight of the relationship between web service and SOAP, SOAP performance measures and evaluation tool, bottlenecks facing SOAP, techniques used in overcoming these bottlenecks and the developed hybrid architecture. We then implemented the hybrid architecture, set-up experiments, and collected data, evaluated, and subsequently discussed research's results. SOAP web services are implemented in ASP.NET. We then examine SOAP on HTTP in a local area network.

1.7 STRUCTURE OF THESIS

The rest of this research is divided into several chapters. Chapter two discusses literature review. Chapter three outlines software development. Chapter four covers experiment data collection, results and discussion. Chapter five gives summary, conclusions and recommendations.

CHAPTER TWO

LITERATURE REVIEW

2.1 SERVICE-ORIENTED ARCHITECTURE

A service is an implementation of well-defined business functionality. These services can then be consumed by clients in different applications or business processes (Qusay, 2005).

Service-Oriented Architecture (SOA) can be defined in many ways:

- SOA is an architectural style for building software applications that use services available in a network such as the web (Qusay, 2005).
- SOA is a loosely-coupled architecture designed to meet the business needs of the organization (Linthicum, 2013).
- SOA is an IT architectural style that supports the transformation of your business into a set of linked services, or repeatable business tasks that can be accessed when needed over a network (IBM, 2013).

SOA is a very popular paradigm in developing distributed systems. SOA is flexible, extensible and robust in the sense that it allows for the reuse of existing assets where new services can be created from an existing IT infrastructure of systems. This enables businesses to leverage existing investments by allowing them to reuse existing applications. SOA promises interoperability between heterogeneous applications and technologies (Qusay, 2005). Service-oriented applications expose business logic via services.

Internet growth is immense and continuous. It has become an important tool in communication, providing services and sharing information. Organizations are embracing the internet in many ways e.g. e-commerce, blogs among others. Such services can be provided in the internet by using SOA, WOA etc. in software applications. SOA is emerging as the premier integration and architecture framework in today's complex and heterogeneous computing environment (Qusay, 2005). SOA is being adopted by many programmers as a way integrating software systems and providing different services thus building dynamic systems that are of loosely coupled architecture. Technologies that implement SOA include: CORBA, Java RMI, COM and web services.

There has been a paradigm shift from object oriented analysis and design in the 80’s to component-based development design in the 90’s and now we have service-oriented architecture. This is a transformation from remotely invocation of objects to message passing between services. Schemas in service-oriented describe not only the structure of messages, but also behavioral contracts to define acceptable message exchange patterns and policies to define service semantics. These characteristics promote interoperability and thus provide adaptable benefits. This implies that a message sent from one service to another is done without the programmer’s consideration of how the service handling these messages has been implemented (Linthicum, 2013).

2.2 ROLES OF A SERVICE-ORIENTED ARCHITECTURE

The roles of a SOA, as shown in Figure 2.1, are as follows (Endrei et al., 2004):

(i) Service consumer: This is an application, a software module or another service that requires a service. It initiates the enquiry of the service in the registry, binds to the service over a transport, and executes the service function. The service consumer executes the service according to the interface contract.

(ii) Service provider: This is a network-addressable entity that accepts and executes requests from consumers. It publishes its services and interface contract to the service registry so that the service consumer can discover and access the service.

(iii) Service registry: This is the enabler for service discovery. It contains a repository of available services and allows for the lookup of service provider interfaces to interested service consumers.

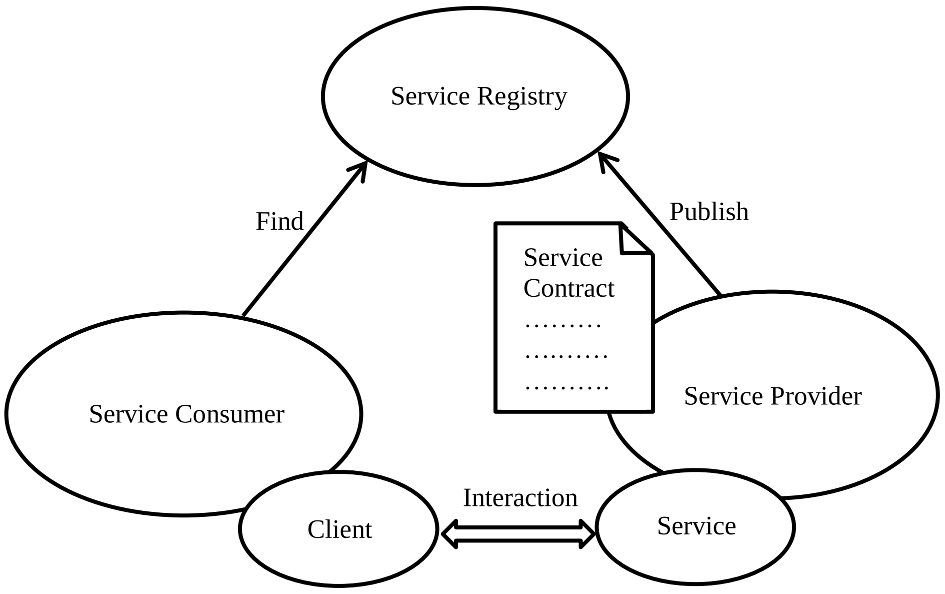


Figure 2.1: Roles of a service-oriented architecture

Adopted from Snell et al. (2001)

2.3 OPERATIONS IN A SERVICE-ORIENTED ARCHITECTURE

The operations in a service-oriented architecture are as follows (Endrei et al., 2004):

(i) Publish: This makes a service accessible. A service description must be published so that it can be discovered and invoked by a service consumer.

(ii) Find: A service requestor locates a service by querying the service registry for a service that meets its criteria.

(iii) Bind and invoke: After retrieving the service description, the service consumer proceeds to invoke the service according to the information in the service description.

The artifacts in a service-oriented architecture are as follows (Endrei et al., 2004):

(i) Service: A service that is made available for use through a published interface that allows it to be invoked by the service consumer.

(ii) Service description: A service description specifies the way a service consumer will interact with the service provider. It specifies the format of the request and response from the service. This description may specify a set of preconditions, post conditions and/or quality of service (QoS) levels.

Some of SOA characteristics are: its self-contained (can be deployed independently), it is a distributed component (its services are available over the network and can be accessible through a name or locator other than an absolute internet address), it has a published interface (users of the service only need to see the interface and can be oblivious to implementation details), interoperability in heterogeneous systems (service users and providers can use different implementation languages and platforms), it is discoverable (has a special directory service that allows the service to be registered and users can look for services) and dynamically bound (the service is located and bound at runtime) (Bianco et al., 2007).

Web service is one of the standard-based techniques to realize SOA. There are various advantages and disadvantages of adopting CORBA, Java RMI, COM or web services in implementing SOA. Web service stands out amongst its equals who are monolithic in nature. Moreover, web service is platform independent, can be integrated in various heterogeneous applications, has the capability of riding on HTTP and can go through firewalls without being recognized.

2.3.1 WEB SERVICE IN SERVICE-ORIENTED ARCHITECTURE

Web services can be defined in various ways:

- Web services extend the World Wide Web infrastructure to provide the means for software to connect to other software applications. Applications access Web services via ubiquitous web protocols and data formats such as HTTP, XML, and SOAP, with no need to worry about how each Web service is implemented (Microsoft, 2013).
- A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a format (specifically WSDL) that can be processed by a machine (Booth et al., 2004).
- A web service is a set of related application functions that can be programmatically invoked over the Internet. Businesses can dynamically mix and match web services to perform complex transactions with minimal programming. Web services allow buyers and sellers all over the world to discover each other, connect dynamically and execute transactions in real time with minimal human interaction (IBM, 2005).

The roles of web services in the web service architecture complement those of SOA which include: service provider, service requestor/ consumer and service registry. Web services provide a distributed computing approach to enable interoperability between heterogeneous (different software applications, running on different platforms and/or frameworks) systems. In the internet web services specifications are completely independent of programming language, operating systems and hardware to promote loose coupling between service consumer and service provider (Mark et al., 2004).

Figure 2.2 shows a Venn diagram. The Venn diagram shows how web services encapsulate cloud computing technologies, how web services technology are part of SOA and vice versa and how cloud computing can be part of SOA and vice versa. Cloud computing uses web services for connectivity (Barry & Associates, 2013).

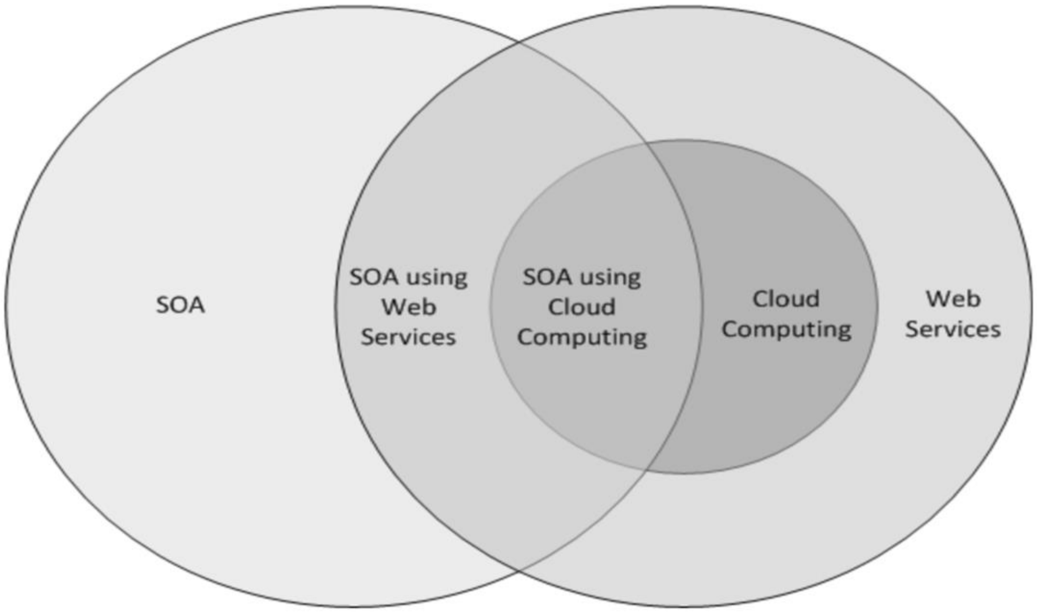


Figure 2.2: Relationship between SOA, cloud computing and web services

(Adopted from Girish et al. (2013))

In pure web services solution architectures, SOAP protocol is widely used. Another approach is to use Representational State Transfer (REST). One more option is to use messaging systems, such as Microsoft MSMQ and IBM WebSphere MQ (previously called MQSeries).

Web services take the ideas and principles of the web and apply them to computer interactions. Like the World Wide Web (www), web services communicate using a set of foundation protocols that share a common architecture and are meant to be realized in a variety of independently developed and deployed systems. Nonetheless, like the www, web services protocols owe much to the text-based heritage of the internet and are designed to layer as cleanly as possible without undue dependencies within the protocol stack, as shown in Figure 2.3.

2.3.2 WEB SERVICES PROTOCOL STACK

Snell et al. (2001) elaborate web services stack as shown in Figure 2.3.

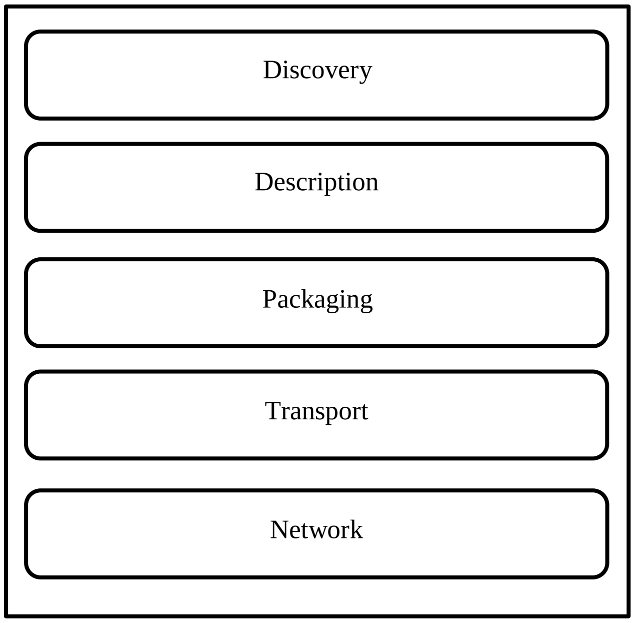


Figure 2.3: The web service technology stack

(Adopted from Snell et al. (2001))

The web service technology stack has five layers. These layers perform different functions which are discussed as follows:

(i) Discovery

Discovery is the act of locating resource description. It is performed by the end user and is realized using a discovery service. The resources being discovered are usually service descriptions thus enabling centralizing services into a common registry and providing easy publish/find functionality. Currently service discovery is handled by Universal Description Discovery and Integration (UDDI).

(ii) Description

This layer is responsible for describing the public interface to a specific web service. Currently, service description is handled by Web Service Description Language (WSDL). Other approaches include W3C's Resource Description Framework (RDF) and DARPA Agent Markup Language (DAML) which provide a richer capability but very complex to describe web services than WSDL. Two attributes in WSDL interface include: style and use. The style attribute has two possible values: RPC or document while the use attribute has two possible values: encoded or literal. The style and use attributes form four possible combinations called binding styles, common once being RCP-encode and document-literal. Bianco et al. (2007) noted that RPC-encode have more overheads than document-literal. SOAP binding styles will be covered in detail in section 2.5.2.

(iii)Packaging

Packaging must be done before the message is relayed in the network and must be in a format that all parties in heterogeneous environment understand. HTML is primarily used in SOAP communication because it is text-based and can be implemented using a variety of operating systems, programming environments and is tied to representation other than meaning. XML is the basis for web services packaging formats because it can be used to represent the meaning of the data being transferred. Moreover, XML parsers are now ubiquitous. SOAP does the packaging in either RPC-encoded or document style. SOAP is a standardized de facto packaging protocol.

(iv)Transport

This layer has several technologies that enable application-to-application communication on top of the network layer. These technologies include: TCP, HTTP, SMTP and Jabber. The choice of these technologies is as per the needs of web services implementation. This layer transports data in the network. HTTP is one important protocol to note because of its platform independence and provides most ubiquitous firewall support. However, HTTP does not provide support for asynchronous communication. Jabber on the other hand provides a good communication channel.

(v) Network

Network in web services architectural stack is like the network layer in TCP/IP Network Model which provide critical basic communication, addressing and routing capabilities.

In summary, web service was developer of open technologies. These open technologies provide an opportunity for interoperability among solutions from different vendors' i.e. heterogeneous platforms. Therefore, organizations can implement web services without having knowledge of the service consumers and vice versa; enabling a just-in-time integration and allowing businesses to establish new partnership easily and dynamically. The enabling technology in web services is XML protocols. XML protocols govern how communication happens and how data is represented in XML format in the wire. Web service technology consists of several publishing standards, SOAP and WSDL.

2.4 SOAP IMPLEMENTATION

2.4.1 OVERVIEW OF SOAP

SOAP is a standardized de facto XML-based protocol for packaging, services invocation and exchanging messages in distributed systems aided by web service interfaces. The specification defines nothing more than a simple XML-based envelope for the information being transferred, and a set of rules for translating application and platform-specific data types into XML representations (Snell et al., 2001).

SOAP is a flexible and highly extensible protocol since it is XML based. SOAP is aimed at low-level distributed computing system developers. This implies that in distributed computing SOAP can be used to enable interoperability of remote applications. SOAP works behind the scenes (abstraction) making sure that your applications can interoperate without much programming effort. SOAP is ubiquitous and most if not all software vendors (e.g. Java, IBM, Microsoft etc.) have embraced and integrated it in their programming environments. This has made SOAP to be a highly abstract and flexible technology.

In XML messaging, applications exchange messages using XML documents. This forms a flexible way of communication and forms the basis of using SOAP. A message can be anything e.g. purchases order, report of sales records etc. XML is not tied to a particular operating system or programming language; this makes it usable to all environments. For example, Java programs can create an XML document which can be shared in Perl programs, Microsoft programs etc. The fundamental idea is to share information using simple messages encoded in a way both applications can understand. SOAP provides a simple way to structure messages (Snell et al., 2001).

2.4.2 SOAP MESSAGE STRUCTURE

The structure is as described in Figure 2.4

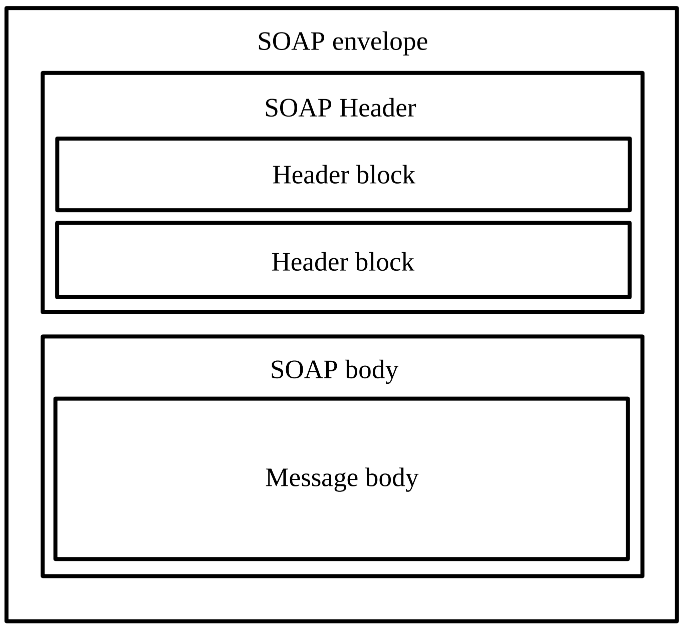


Figure 2.4: SOAP message structure

(Adopted from Papazoglou (2008))

SOAP message structure has four regions:

(i) SOAP envelope

The SOAP <Envelope> is the root element in every SOAP message, and contains two child elements, an optional <Header> and a mandatory <Body>

(ii) SOAP header

The SOAP <Header> is an optional sub-element of the SOAP envelope, and is used to pass application-related information that is to be processed by SOAP nodes along the message path.

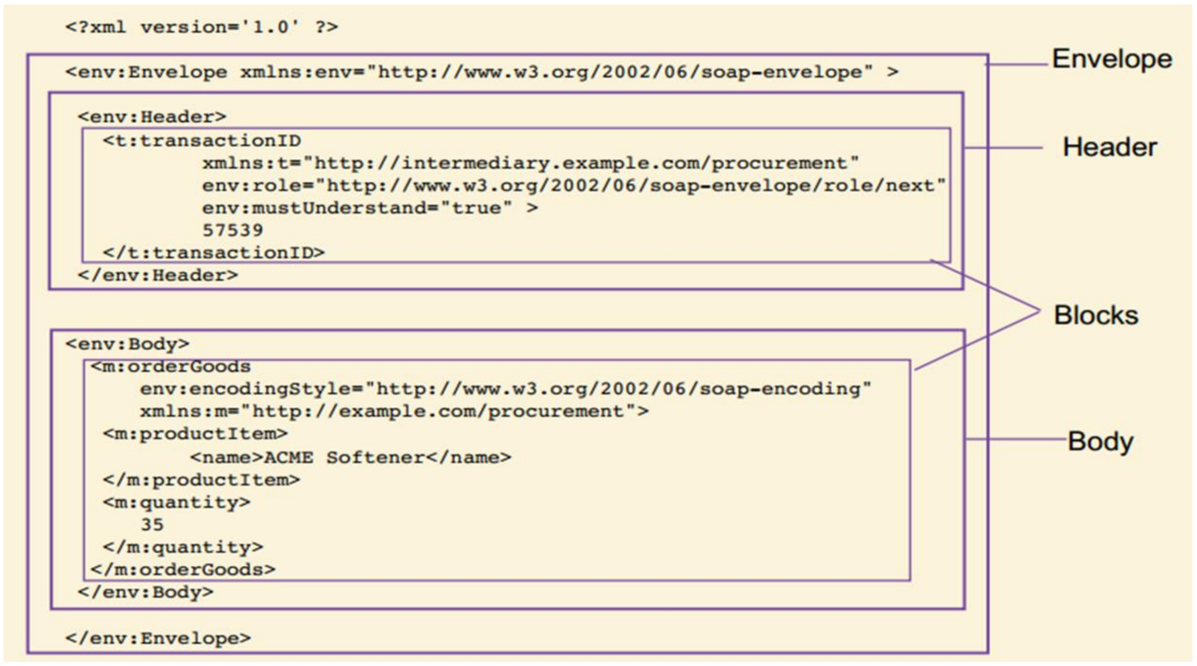
(iii) SOAP body

The SOAP <Body> is a mandatory sub-element of the SOAP envelope, which contains information intended for the ultimate recipient of the message.

(iv) SOAP fault

The SOAP <Fault> is a sub-element of the SOAP body, which is used for reporting errors.

An example of a SOAP message is as shown in Figure 2.5.



```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
 <env:Header>
  <t:transactionID
          xmlns:t="http://intermediary.example.com/procurement"
          env:role="http://www.w3.org/2002/06/soap-envelope/role/next"
          env:mustUnderstand="true" >
          57539
  </t:transactionID>
 </env:Header>

<env:Body>
  <m:orderGoods
     env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"
     xmlns:m="http://example.com/procurement">
   <m:productItem>
         <name>ACME Softener</name>
   </m:productItem>
   <m:quantity>
     35
   </m:quantity>
  </m:orderGoods>
 </env:Body>

<env:Envelope>
```

Figure 2.5: An example of a SOAP Message

(Adopted from Papazoglou (2008))

2.4.3 SOAP BINDING STYLES

WSDL attributes was briefly described in 2.3.2 where this research identified two combinations of style and use attributes in WSDL. This affects the performance of SOAP as follows:

(i) RPC-encoded style

This is an equivalent to an XML-based remote method call where the name of the method and type of argument are specified in the WSDL interface definition. Results are returned to the calling method (Snell et al., 2001). The attributes are encoded using a standard encoding format. RPC-encoding style was popular in the first few years when web service was introduced because of its simple programming model and the similarity between services and object methods. However, it has its own short-comings because of the deficiencies of SOAP-encoding specifications (Bianco et al., 2007). Figure 2.6 shows an example of RPC-encoding call.

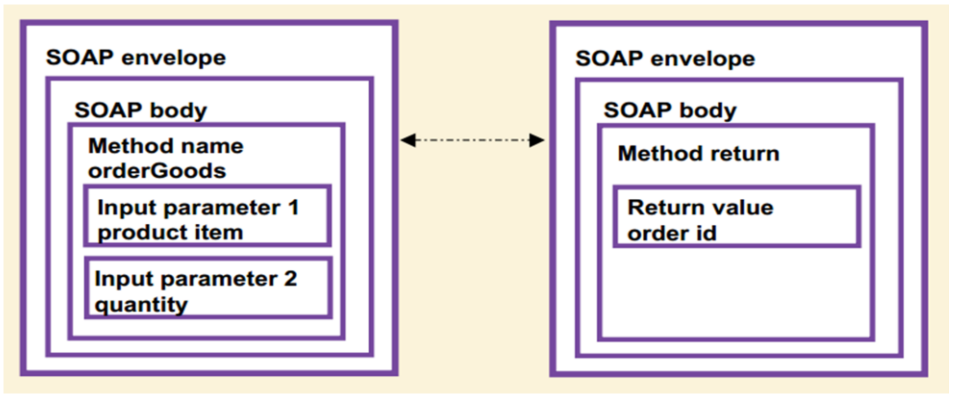


Figure 2.6: SOAP RPC-encoding call

(Adopted from Papazoglou (2008))

(ii) Document-literal style

It is at times called Electronic Data Interchange (EDI). The SOAP messages body in the document-literal style contains an arbitrary XML (business document). With this approach there are no parameters specified as the RPC-style. The literal attribute indicates that no standard encoding format is used; data in the SOAP body is formatted and interpreted using the rules specified in XML schemas created by the service developer. The XML schemas that define the data structure of the request and the response are the key elements in the interface definition (Bianco et al., 2007). Figure 2.7 shows an example of a document/literal call.

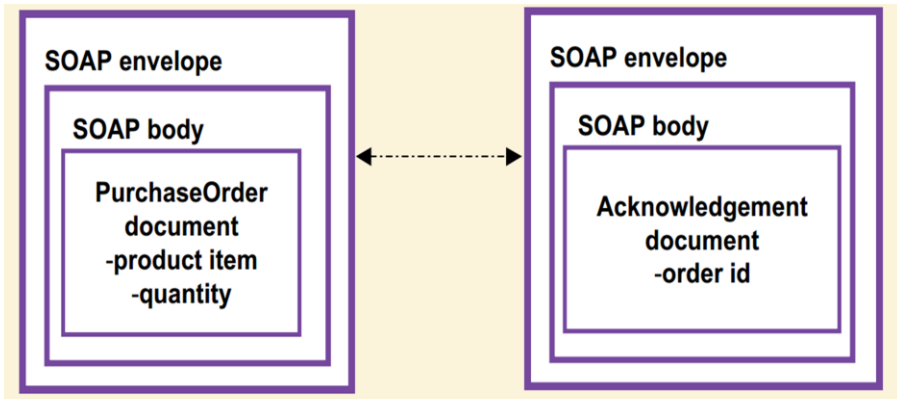


Figure 2.7: SOAP document-literal call

(Adopted from Papazoglou (2008))

In summary, Bianco et al. (2007) recommended the use of document-literal style approach because it has lesser overheads as compared to RPC-encoded style. Some web services use RPC-encoding style, it is important to note these specifications before creating a web service.

2.4.4 SOAP OVER HTTP

SOAP protocol relies on HTTP or HTTPS in communication. However, SOAP can also ride on SMTP and other compatible network transfer/ communication protocols. The advantage of riding on HTTP is that, HTTP is: firewall friendly, an open standard and a universally accepted transfer protocol. SOAP messages are encapsulated within HTTP. HTTP is a universal standard in the WWW. Firewalls by default allow traffic through port 80 which HTTP uses in communication. This gives SOAP power to be platform independent (Papazoglou, 2008). SOAP does requests and responses via HTTP. SOAP uses the HTTP GET method for requests and HTTP POST method for both request and response, as shown in Figure 2.8. HTTP explores TCP/IP protocol for network transport because of reliability nature of TCP/IP protocol. Some researchers have explored HTTP binding on UDP which proved to improve performance but it was less reliable (Kho, 2007). In equation (2.1) we can simply say that:

$$\text{HTTP} + \text{XML} = \text{SOAP} \tag{2.1}$$

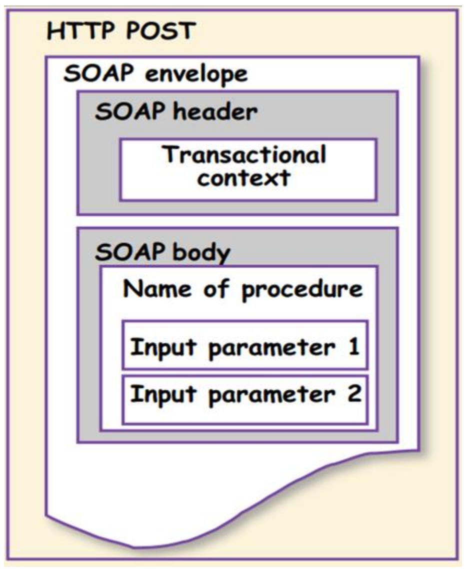


Figure 2.8: HTTP SOAP binding

(Adopted from Papazoglou (2008))

Figure 2.9 shows SOAP binding with HTTP and how messaging occurs in RPC-style.

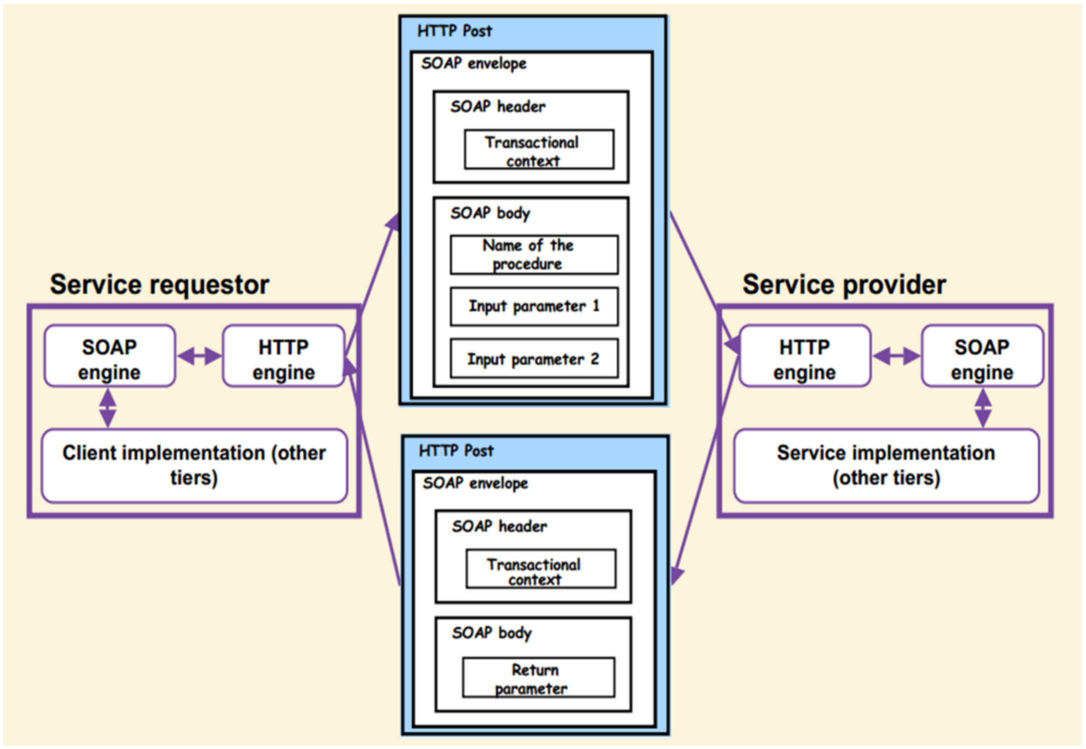


Figure 2.9: SOAP binding in HTTP and communication using RPC-style.

(Adopted from Papazoglo (2008))

2.5 SOAP PERFORMANCE IMPROVEMENT TECHNIQUES

Tekli et al. (2011) provides concise and comprehensive research efforts aimed at SOAP performance enhancement (Tekli et al., 2011). They provide a unified view of the problem that covers almost every phase of SOAP processing including: message parsing, serialization, deserialization, compression, multicasting, security evaluation, and data/instruction-level processing. Nevertheless, other techniques have been realized that Tekli et al. (2011) did not highlight. This research has identified and classified some of the SOAP improvement techniques thematically as: client-side, communication channel and server-side.

2.5.1 CLIENT-SIDE TECHNIQUES

A client is a computer that sends requests to the server; normally it is the end users computer. All the computing operations involved in client computer are said to exist in the client-side. In this section we discuss client-side caching technique and differential serialization (DS).

(i) Client-side caching

Client-side caching is the storage of data in the client-side. Client-side caching is a technique of improving SOAP performance through improved response time. SOAP client-side caching has been supported by several researchers (Seyyed et al., 2011) (Chandra et al., 2012) (Kiran et al., 2003) (Hou et al., 2010) (Matthieu et al., 2009). Caching has been embraced solely to improve the amount of traffic and latency between the service and underlying data providers (Seyyed et al., 2011) (Chandra et al., 2012) (Matthieu et al., 2009). Client caching can store data temporarily within the internet browser or by a JavaScript data structure (Matthieu et al., 2009).

Data in SOA is categorized as service state and service result. Service state is data that concerns the state of the business process/service while service result is data that is delivered by the business process/service back to the presentation layer. Moreover, caching can be categorized as: client-side caching, proxy caching, reverse proxy caching, and web server caching. Figure 2.10 shows the different types of caching.

In client-side caching, data is stored by the client-side browser temporarily on the local disk or browser’s internal memory. Its advantage is that, data cached on the local client can be easily accessed thus reducing network traffic while its disadvantage is that, cached data in the client is browser dependent and not shareable. Proxy caching uses a proxy server that stores cached data between the client and the web server. This data cached in proxy server can be shared among clients thus leveraging the weakness identified in client-side caching. Its advantage is that it fulfills all requests from web page without sending them out to the actual web server over the internet, resulting in faster access and reduced traffic. Its disadvantages include deployment and infrastructure overhead to maintain the proxy servers. In reverse proxy caching, the proxy is placed in front of the web server. The proxy responds to the most frequent request and passes others to the web server. As much it reduces the number of requests directed towards the web server, its position in front of the server increases network traffic. In web server caching, the web server stores its own cached data. It improves the performance of a site by decreasing the round trip of data retrieved from database or other servers, reduces server load, and reduces bandwidth consumption (Chandra et al., 2012)

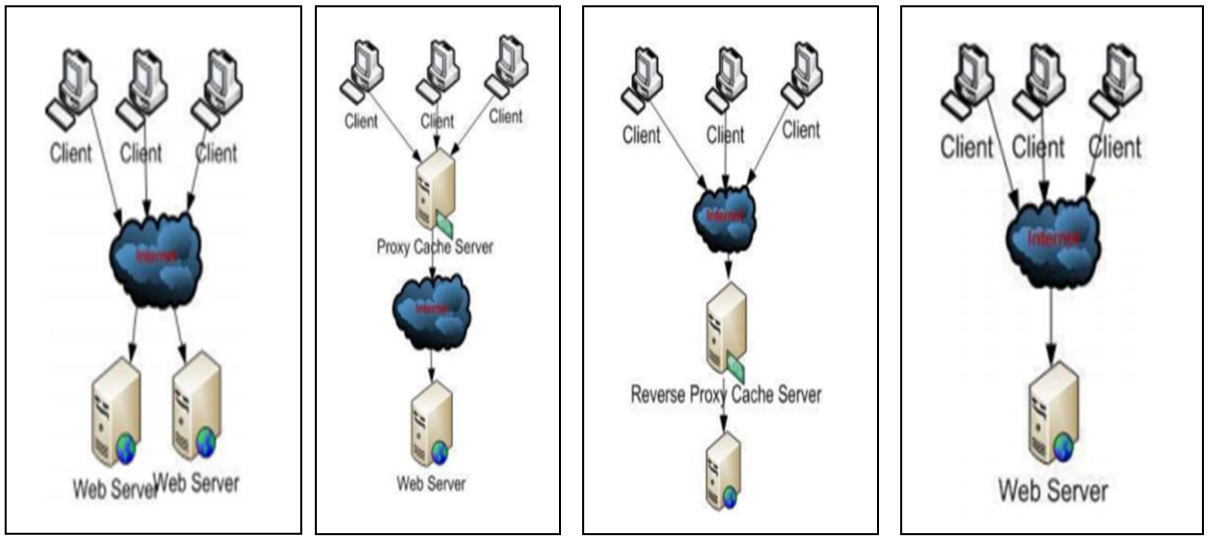


Figure 2.10: Client-side caching, proxy caching, reverse proxy caching and web server caching respectively

(Adopted from Chandra et al. (2012))

After profiling the client-side Kiran et al. (2003) noted that around 40% of execution time is spent in XML encoding which involves serializing and marshalling the SOAP payload before transmitting it to the server. Chandra et al. (2012) and Matthieu et al. (2009) noted an improved performance using client-side caching. Therefore, clients that send the same request to the server frequently consume a considerable amount of time in encoding XML. To overcome this challenge, caching such request(s) not only saves a considerable amount of execution time in recreating the payload, but also the time involved in trips to fetch data from the server. The client always checks if the request was previously indexed and cached on the client-side before sending it to the server. If the request was cached, it does a simple file I/O operation to fetch the payload from the client-side cache (Kiran et al., 2003). Kiran et al. (2003) used RPC-style in WSDL 1.1 binding which is an inefficient SOAP binding style as will be covered in section 2.5.2. However, after an evaluation of SOAP caching on the client-side, Kiran et al. (2003), recorded an improved performance by a remarkable 800%. This resulted to better performance than the traditional binary Java RMI which outperformed SOAP as discussed in (Davis et al., 2002). Figure 2.11 shows the round trip results of SOAP with Java RMI.

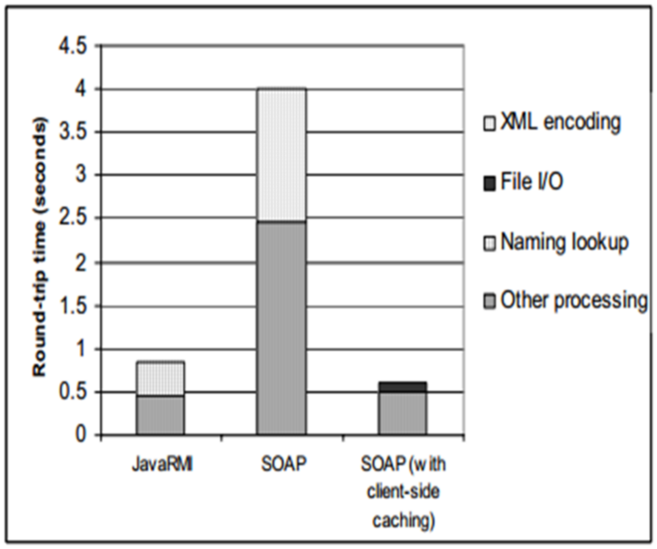


Figure 2.11: Comparison of SOAP (with client-side caching) with Java RMI and the traditional SOAP

(Adopted from Kiran et al. (2003))

Some of the challenges involved in client-side caching include: how frequent the data needs to be updated, the data being user specific or application-wide, and what mechanism to use to indicate that the cache needs updating (Chandra et al., 2012) (Kiran et al., 2003) (Matthieu et al., 2009) (Junichi et al., 2005). Matthieu et al. (2003) and Kiran et al. (2003) noted that proper indexing and time stamping can be used to verify its validity. Consequently, Kiran et al. (2003) notes that data in the client-side can be updated not only by deleting and renewing data period of time, but also by updating last modified timestamps by use of a cache provider. Updating last modified timestamps is much better because it imposes less overhead as compared to reloading the entire data set. Matthieu et al. (2003) suggested a hybrid reverse caching strategy in web caching. Hybrid reverse caching caches data structures rather than static values. This caching can be built on unified data stores to eliminate redundant and duplicate data. Client-side caching performance can be enhanced further by doing more research on caching algorithms that can further improve fetching and serialization of XML data.

(ii) Differential Serialization

Differential serialization (DS) avoids serializing of the whole message structure. Serialization of sent/outgoing messages involves conversion of in-memory data types to SOAP XML-based ASCII string formats, and then packing this data into message buffer; this counts as one of the major performance bottlenecks of SOAP performance as it accounts for 90% of end-to-end message time. The client that sends the requests is called the bSOAP. In DS, once a serialized message has been sent by a SOAP communication end point, the client saves the message so that it can be reused by future subsequent messages as a template. Subsequent messages that have the same structure or are identical can reuse the structure and avoid the serialization overhead involved in regenerating the structures from scratch. This works best if the same client sends a stream of similar messages. This technique improves response time (Seyyed et al., 2011) (Dhiah et al., 2010) (Nayef et al., 2004) (Behrouz et al., 2009) (Nayef et al., 2005) (Tekli et al., 2011).

The steps to follow to make DS a success is: tracking data changes and overwriting only those values that have been changed since the last sent message, expanding the serialized message to accommodate large serialized values, storing the message in chunks and padding them with white spaces to reduce the cost of expansion, and overlaying the same memory region with different portions of the same outgoing message to reduce memory consumption. The steps outlined above demonstrated a best case performance of ten times faster. The study also showed that send times reduced by a factor of five only when parts of the message were to be re-serialized. In designing the DS, when comparing the outgoing message to the saved templates the different matching possibilities are (Nayef et al., 2004):

- Message content matching: This entails the entirely sent message being exactly the same as the one sent from the client earlier; the client sends the message as it is.
- Perfect structure match: This entails the message having the same structure and size as an earlier message but having values of some field that have changed. In this case, the serialized message is replaced with the changed values only.
- Partial structure match: This entails the message having a structure but a change in size of the message as compared to an earlier message. Also, some of the values may not have matched. Unlike in memory base types, the serialized message template may be expanded or contracted to meet the requirements of the new message.
- First time send: This phase encounters the normal overheads involved in creating a serialized message from scratch, checking whether it exists amongst the saved templates and saving a pointer to it after it has been created.

From the different matching possibilities, researchers in (Nayef et al., 2004) notes that partial structure match can be avoided using several techniques which include: stuffing, shifting, chunking, and stealing.

- Shifting: This involves expanding the message in memory when the serialized form of a new message exceeds its field width as shown in Figure 2.12. It involves shifting bytes in the template to make room for the new values then updating Data Update Tracking (DUT) Table accordingly. This is expensive because it entails memory moves, possibly memory reallocation, and updating DUT table.

```
…</w><x xsi:type='xsd:int'>1.2</x><y xsi:type=….
becomes
…</w><x xsi:type='xsd:int'>1.23456</x><y xsi:type=….
```

Figure 2.12: Shifting technique in Differential Serialization.

(Adopted from Nayef et al. (2004))

- Stuffing: This involves adding extra white spaces in the serialized message to accommodate potential future updates that would otherwise require expansion as shown in Figure 2.13. The white spaces can be explicitly created when the template is created or after serializing a value that requires less space. This technique can avoid shifting which is an expensive technique.

```
…<y xsi:type='xsd:int'>678</y><z xsi:type=…
can be represented as
…<y xsi:type='xsd:int'>678</y>□□□□<z xsi:type=…
```

Figure 2.13: Stuffing technique in Differential Serialization

(Adopted from Nayef et al. (2004))

- Stealing: This reduces the costs of increasing field size by stealing extra spaces from neighboring fields instead of shifting entire portions of memory chunks. This technique is actually less expensive than shifting. Performance of stealing depends upon the Halting Criteria (tell when to stop stealing) and direction (tell left, right or back-and-forth of memory chunks).

```
…'>678</y><z xsi:type='xsd:double'>1.166</val>□□□□□
y can steal from z to yield…
…'>677.345</y><z xsi:type='xsd:double'>1.166</val>□
```

Figure 2.14: Stealing technique in Differential Serialization

(Adopted from Nayef et al. (2004))

- Chunking: This involves storing messages in potential non-contiguous memory chunks to limit the impact of the expensive Shifting.

Clients that send the same message frequently can maximize the advantage of DS in improving performance of that system. DUT Table comes in handy to track whether a program has changed data items in new messages since the last serialized SOAP messages. DS had an impact of up to 17% improvement (Nayef et al., 2004). However, in as much as Chiu et al. (2002) proposed an optimized version called XSOAP that used a new XML parser specialized for SOAP arrays, Suzumura et al. (2005) notes that, in scientific grid computing (an area of high performance computing that is adopting the web service architecture), sending scientific data e.g. large arrays of floating point numbers and complex data types via standard implementation of SOAP is very expensive.

2.5.2 COMMUNICATION CHANNEL TECHNIQUES

The SOAP XML document is embedded in HTTP as the default transport protocol. SOAP messages can be transported in SMTP and FTP among other protocols. HTTP uses port 80 as the default communication port. By default SOAP uses HTTP-GET or HTTP-POST protocol to communicate in WS (Papazoglou, 2008) (Kho, 2007). The wire format of data in communication channel affects SOAP performance (Kiran et al., 2003) (Fabian et al., 2000). This section discuss SOAP binding style and compression techniques that can improve SOAP communication

(i) SOAP Binding Style

In section 2.3.2, we covered web service stack which contains the description layer. The description layer is responsible for describing the public interface of a specific web services. Services are exposed on the public interface of a web service. Service description (location and methods exposed) is handled by Web Service Description Language (WSDL). Other approaches include the W3C's Resource Description Framework (RDF) and DARPA Agent Markup Language (DAML) which provide a much rich capability but very complex to describe web services than WSDL (Bianco et al., 2007).

WSDL is a model that provides an XML format for describing WS in the web community; this ensures interoperability in heterogeneous systems. As per WSDL 1.1 (Erik et al., 2001) standards, the document structure of the XML has two sections abstract and concrete. The abstract elements (Type, Message, and PortType) define WS interface while the concrete section (Binding and Service) describes how abstract interface maps messages on the wire (Bianco et al., 2007) (Erik et al., 2001) (Aaron, 2003). All the WSDL 1.1 elements include:

- Type: This is a container for the schema type definitions.

- Message: This defines an abstract message that serves as the input/output of an operation. An operation is a message exchange; a focal point of a service interaction.
- PortTypes: It is also known as Interfaces. It is an abstract set underpinned by one or more endpoints. It describes a function signature (operation name, input parameters, and output parameters) in a Message. An endpoint defines a combination of an address and a binding e.g. URI.
- Bindings: This is a concrete protocol and data format specification for a particular PortType.
- Services: This is a collection of related network endpoints. An endpoint is a port.

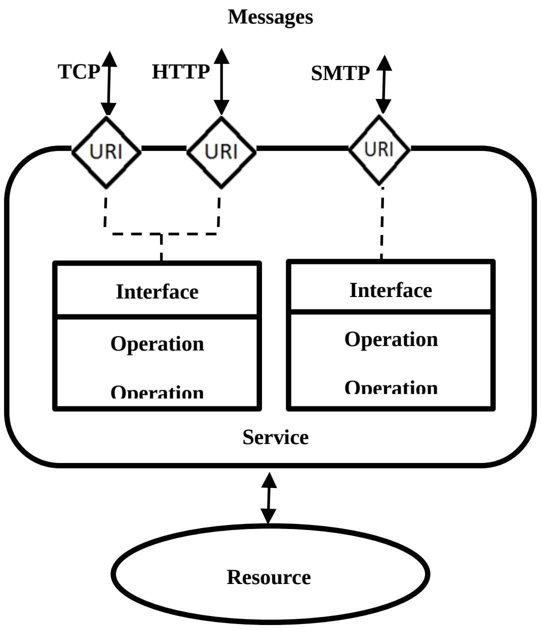


Figure 2.15: WSDL Interface and bindings

(Adopted from Bianco et al. (2007))

This research is interested in the binding element. Binding defines the message format and protocol details for operations and messages as defined by a particular PortType. In WSDL 1.1, binding has two attributes which include: style and use. The default style of the service is either

RPC or document and the default transport protocol (HTTP) while in the communication channel (Erik et al., 2001). The styles are discussed as follows:

- Document-style (previously called message-style) in SOAP dictates that the body contains an XML document, and the message part specifies the XML elements.
- RPC style in SOAP dictates that the body contains an XML representation of a remote procedure being invoked and the message parts representing the parameters to the method.

The use attribute specify the encoding to be used to translate the abstract message parts to concrete representations. It has two possible values of encoded or literal (Bianco et al., 2007).

- In encoded, abstract definitions are translated to a concrete format by using the SOAP encoding rules.
- In literal, the abstract type definitions turn to be the concrete definitions, that is, you can simply inspect the XML Schema type definitions to validate concrete message format.

The style and use attributes forms four possible combinations called binding styles, common once being document-literal as shown in Figure 2.16 and RPC-encode and as shown in Figure 2.17.

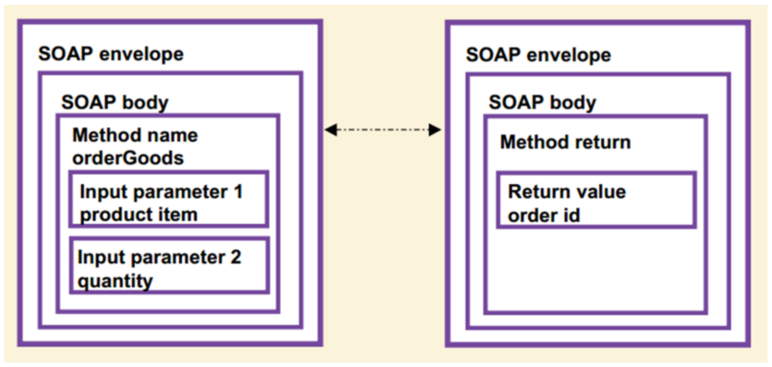


Figure 2.16: SOAP document-literal call

(Adopted from Papazoglou (2008))

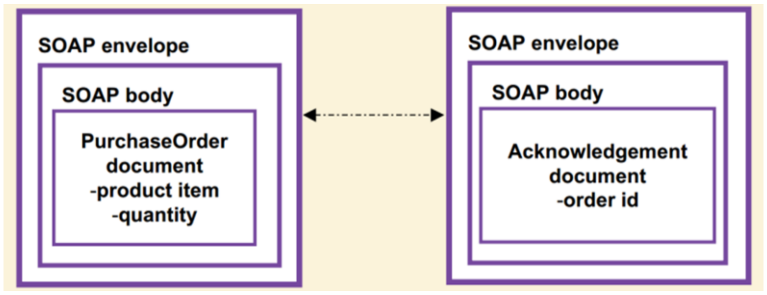


Figure 2.17: SOAP RPC-encoding call

(Adopted from Papazoglou (2008))

A lot of research has been done on the different binding styles and their effects on performance on SOAP in communication. RPC-encode have more overheads than document-literal (Bianco et al., 2007)**.** As much as document style had its own short comings, Bianco et al. (2007) and IBM (2013) recommended adoption of document-literal over RPC style in a bid to improve performance. Java which had document style (MTOM technology enabled) showed that, web service using RPC style requires 15% more time as compared to document- literal style (Girish et al., 2013). Moreover, Alex et al. (2008) notes that test client-side experiments built on document- literal encoding style was faster than previous implementation using RPC. The research findings in Girish et al. (2013) notes that RPC style requires 15% more than document-literal as shown in Figure 2.18.

WSDL 2.0 (Masoud et al., 2009) is a later version of WSDL 1.1. WSDL 2.0 comes with certain features and language elements changed and expanded. For example the definitions element is renamed to descriptions, portType element is renamed to interface, port element is renamed to endpoint, and message element is discarded. The message element defined RPC (parameter driven) and message (document type) communication. Due to the limited expressive powers of RPC in message element, WSDL 2.0 discards it altogether and simply allows an operation to reference a type (such as an XML schema element) directly (Masoud et al., 2009).

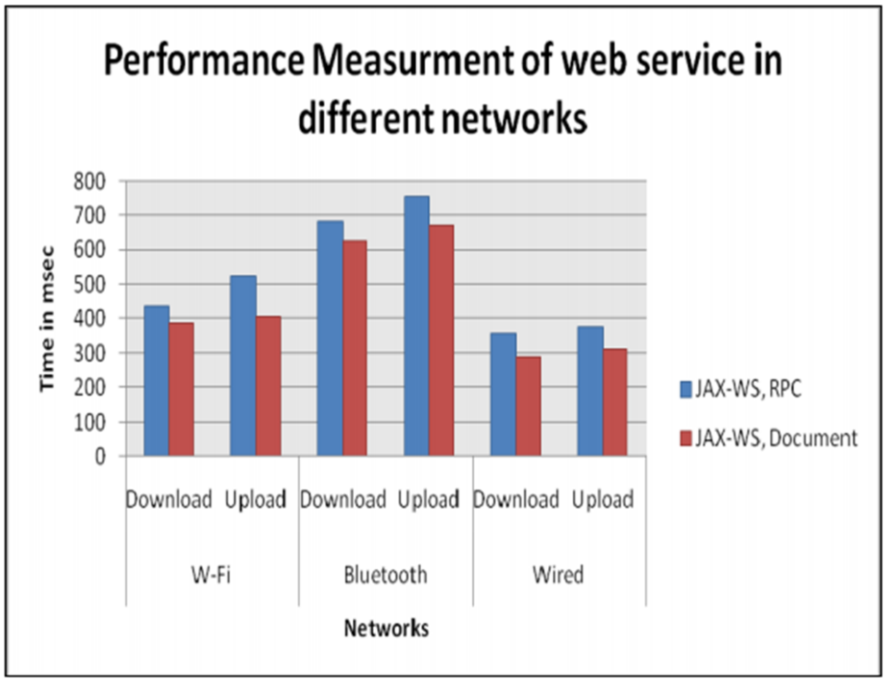


Figure 2.18: Performance measurement of Web service in different networks

(Adopted from Girish, et al. (2013))

WSDL 2.0 component model is a set of components with attached properties which collectively describe a WS. Components in WSDL 2.0 are typed collections of properties that correspond to different aspects of WS. Components in WSDL 2.0 are serializable in XML 1.0 format but are independent of any particular serialization of the component model. WSDL 2.0 components include: description, element, type, interface, interface fault, interface operation, interface message reference, interface fault reference, binding, binding fault, binding operation, binding message reference, binding fault reference, service, endpoint, and extension component (Chiu et al., 2002). WSDL 1.1 the predecessor of WSDL 2.0 has lots of restructuring. Some of the new alterations might be beneficial upon its full understanding (Masoud et al., 2009).

(ii) Compression

A lot of research has been done which supports compression as a promising solution to improving the huge verbose XML messages in SOAP (Seyyed et al., 2011) (Ivan et al., 2008)

(Chandra et al., 2012) (Kiran et al., 2003) (Alex et al., 2008) (Tanakorn et al., 2008) (Tomasz et al., 2010). Compression improves bandwidth utilization and response time of SOAP messages. Compression has its tradeoffs for example extra compression processing time. Kiran et al. (2003) noted that these tradeoffs were not beneficial. However, recently with the increased hardware processing capabilities, these tradeoffs are beneficial as compression is not as costly as increasing bandwidth which is widely under constrains (Ivan et al., 2008).

Different compression algorithms have different compression ratio and different compression time for the same XML file(s) (Hou et al., 2010). An attempt by Kiran et al. (2003) to compact XML tags to reduce the length of the XML tags names had negligible effect on encoding. Kiran et al. (2003) further suggested that other than the data an XML message contained, the major cost of the XML encoding/decoding is in its structural complexity and syntactic elements. Tomasz et al. (2010) and Alex et al. (2008) notes that XML files are highly redundant thus lossless compression algorithm works out best to achieve better compression ratios. Lossless compression algorithm exploits statistical redundancy to represent sender's data more concisely without errors. However, Tomasz et al. (2010) and Alex et al. (2008) note that lossless compression will not work for high entropy (highly disordered) data e.g. already compressed data, random data or encrypted data as it will result in expansion rather than compression. Lossless compression algorithms include Gzip, Bzip2, Fast Infoset (FI), Efficient XML Interchange (EXI) etc. WS-security performance can be an interesting area to explore.

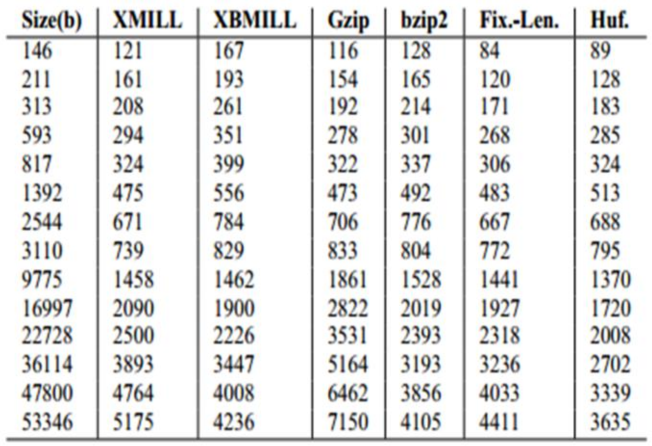
Experiments set up by Tomasz et al. (2010) to compare the best compression algorithm between EXI, FI, and Gzip indicated that FI performed the poorest. EXI showed slightly better compression ratios and response time than Gzip. However, Tomasz et al. (2010) recommended Gzip compression algorithm in disadvantaged network. EXI (Liquid Technologies, 2013) showed promising better performance outcomes than Gzip although it is still under open source test and evaluation (Tomasz et al., 2010). A commercial version of EXI is yet to be released (Liquid Technologies, 2013).

Tanakorn et al. (2008) focused on compression on textual data. They used an algorithm that works in three steps: removal of white spaces, compressing data to UpperCamelCase then decompressing the compressed data. Experiment by Tanakorn et al. (2008) had significant performance gains of up to 22% in bandwidth utilization. The algorithm works in small and large sizes of messages. Nevertheless, experimental results by Tanakorn et al. (2008) show that use of Gzip compression algorithm further improves bandwidth utilization as data integrity is observed. In multimedia data, a detailed analysis of multimedia streaming and compression is tackled in (Ashkan et al., 2012) (Amer et al. 2011). This research is interested in textual data compression.

Nayef et al. (2005) did an evaluation of performance of Gzip and Bzip2 compressors by doing a comparison against three XML compressors (XMILL, xmlppm and XBXML). The methodology involved building an XML tree and converting it into a binary tree then encoding the XML tags by Fixed Length and Huffman techniques. This eventually removes all the closing tags thus saving the opening tags and data leaves of the created tree hence reduces the size of the messages sent and received. Nevertheless, in the experiments, 160 messages were equally divided into four groups in terms of message size categorizes data as small messages (140-800 bytes), medium massages (800-3000 bytes), large messages (3000-20000 bytes), and very large messages (20000-55000 bytes). In the results of XML(uncompressed), XML(Bzip2), XML(Gzip), XMILL(Bzip2), XMILL(Gzip), XMILL(ppm), xmlppm and wbxml, evaluations shows Gzip compression was more effective than Bzip2 by achieving better compression ratio but XMILL(ppm) outperforms Gzip and Bzip2. The findings are shown in Table 2.1. Their experiment conclusions reveal that Huffman encoding was the most efficient for large and very large documents while Fixed Length encoding was found to be efficient for small documents. The compression trends observed in Table 2.1 can be attributed to the fact that look up tables are usually created in lossless compression techniques. Look up tables' aid in mapping of symbols to binary codes during compression process. Therefore, lookup tables in large documents consume a small space as compared to encoded data while in small documents the lookup table tends to be larger than the encoded data. This explains why a high compression ratio is exhibited in large documents as compared to small documents.

Seyyed et al. (2011), Ivan et al. (2008), Tomasz et al. (2010), Tanakorn et al. (2008), Tomasz et al. (2010), Alex et al. (2008) and Kiran et al. (2003) support the fact that compression has a deep impact in not only reducing the response time, but also the improving bandwidth utilization hence increases the performance of SOAP based applications. Seyyed et al. (2011) proposed an architectural design that combined several techniques that improved SOAP performance. Seyyed et al. (2011) proposed architecture is shown in Figure 2.19.

Table 2.1: Result compressed size of different SOAP messages using xmill, xbmill, Gzip, and Bzip2 compressors in addition to fixed and variable length encoding



Size(b)	XMILL	XBMILL	Gzip	bzip2	Fix.-Len.	Huf.
146	121	167	116	128	84	89
211	161	193	154	165	120	128
313	208	261	192	214	171	183
593	294	351	278	301	268	285
817	324	399	322	337	306	324
1392	475	556	473	492	483	513
2544	671	784	706	776	667	688
3110	739	829	833	804	772	795
9775	1458	1462	1861	1528	1441	1370
16997	2090	1900	2822	2019	1927	1720
22728	2500	2226	3531	2393	2318	2008
36114	3893	3447	5164	3193	3236	2702
47800	4764	4008	6462	3856	4033	3339
53346	5175	4236	7150	4105	4411	3635

(Adopted from Nayef et al. (2005))

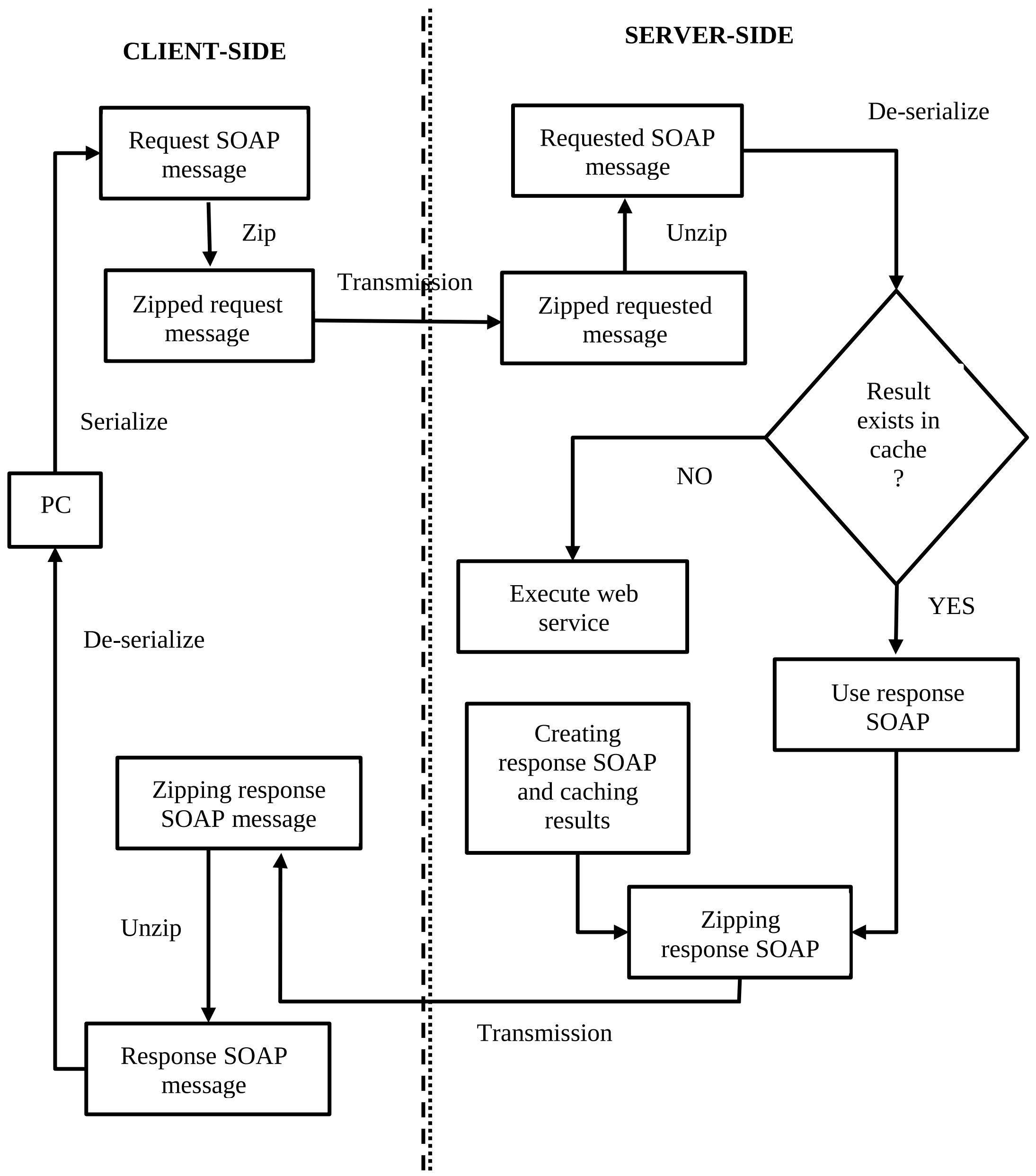


Figure 2.19: Seyyed et al. proposed architecture to improve SOAP performance.

(Adopted from Seyyed et al. (2011))

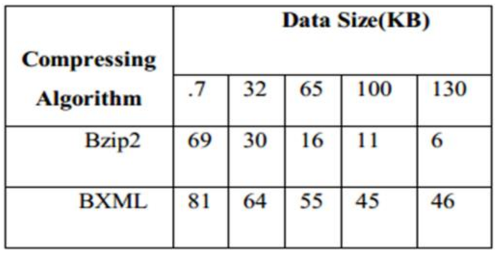
Furthermore, (Hosein et al., 2012) categorized data compression algorithms methods into three:

- General Purpose Compression Algorithm: This include Gzip which is based on Huffman coding, LZ77 which is a substitution compressor and Bzip2 which is an implementation of Burrows-Wheeler block-sorting algorithm

- o XML Aware Compression Algorithm: This explores the separation between XML markup and payload. The simplest in this category are substitution-based algorithms that work at the markup level. They include BXML, WBXML, XMILL, XMLPPM etc.
- o Schema-Aware Compression Algorithm: This defines their schemas in form of XSD or DTD files. They do not encode part of infoset which can be decoded by the receiving party.

In as much as Hosein et al. (2012) argues that compression reduces response time, other factors about a compression algorithm need to be considered e.g. encoding and decoding time, the number of messages transferred, average compression time, number of processes involved, network components passing time, and geographical distance. Experiments by Hosein et al. (2012), prepared in .NET, studies Bzip2 and BXML algorithms considering their response time. From Table 2.2 and Figure 2.20, it is inferred that despite Bzip2 having a better compression ratio than BXML. BXML had a better turn-around time because it had a lower compression time. Moreover, Hosein et al. (2012) notes that when using a compression algorithm, if the data size is more than a specific threshold it may decrease the response time and improve performance otherwise it degrades increases response time and performance.

Table 2.2: The percentage of compression using Bzip2 and BXML algorithms.



Compressing Algorithm	Data Size(KB)				
	.7	32	65	100	130
Bzip2	69	30	16	11	6
BXML	81	64	55	45	46

(Adopted from Hosein et al. (2012))

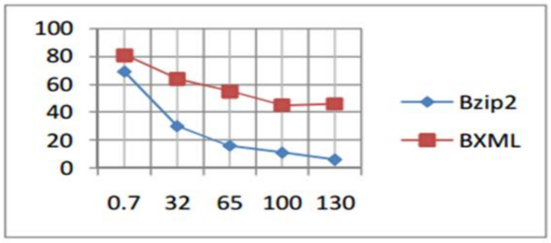


Figure 2.20: The percentage of compression using Bzip2 and BXML algorithms.

(Adopted from Hosein et al. (2012))

Ivan et al. (2008) gave a detailed algorithm on how a client and server communicate exploring text compression technique in a bid to improve performance in web-service-based applications. The algorithm showed text was reduced by 80%, meaning 80% less storage space was saved. Nonetheless, the text data being transferred required less time which translated to high performance for client-server application communication. As much as the compression had some tradeoffs like processing time, it resulted to a general better performance of the system. Figure 2.21 shows an implementation of the compression. Text input from the client through the proxy is serialized as text-based SOAP message, compressed then sent to the server. The text is then decompressed, de-serialized then passed to the web service. The web service processes the request and returns the result which is later serialized and compressed before being sent to the client. Lastly, the client collects the text message which is de-serialized and decompressed through the proxy. The total processing time of a request is given in equation (2.2) where t_{ser} is time needed to serialize the request in xml format, t_{tr} is the time needed to actually transfer the serialized request, t_{deser} is the time needed to de-serialize the xml text and $t_{servproc}$ is the time needed for processing the request and producing the results at the server-side (Ivan et al., 2008).

$$T_{total} = 2\cdot(t_{ser} + t_{tr} + t_{deser}) + t_{servproc} + t_{com/dec} \qquad (2.2)$$

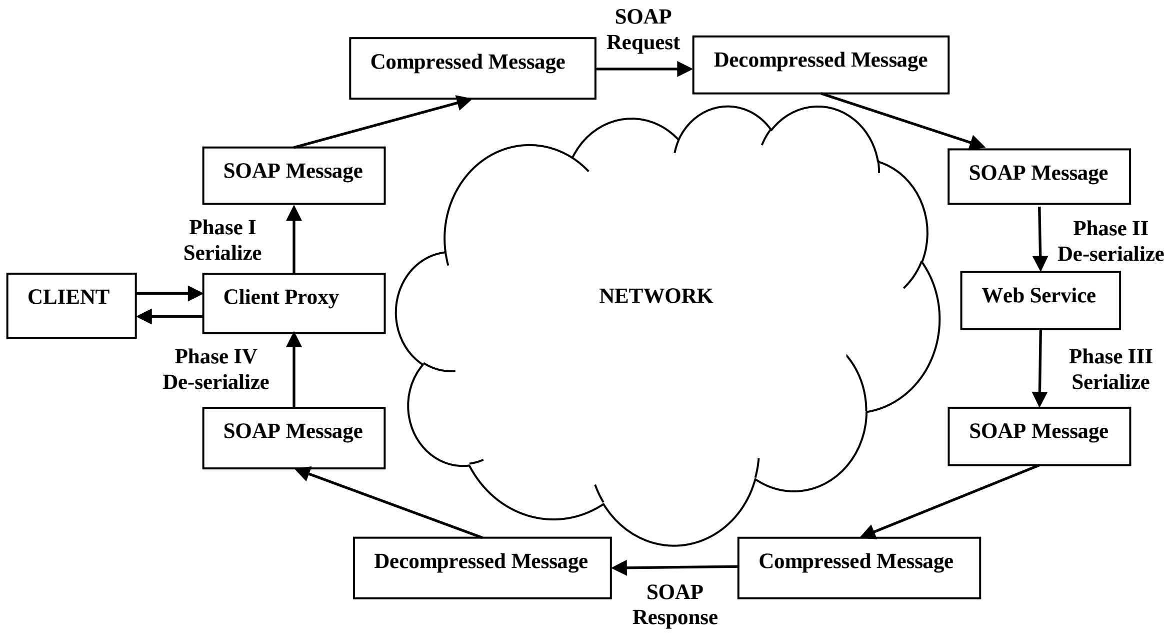


Figure 2.21: Implementation of Compression and Decompression after serialization and before deserialization.

(Adopted from Ivan et al. (2008))

The research done by Hou et al. (2010) gives an assessment formula in equation (2.3) to calculate T which is the transmission time a WS can be improved. In the formulae: N is network speed in bytes per second, C is the computing speed of the device in units per second, compression algorithm requires Z computing units, and compression algorithm can compress the SOAP message out E byte. If the result of T is a positive value then the transmission performance of WS is improved. If it is a negative value, then it means that the compression algorithm does not improve the transmission performance of the algorithm.

In as much as there is a tradeoff between compression and CPU usage, this can be resolved by use of powerful server that uses multi-processing cores (Hazem, et al.). Moreover, Moores's Law clearly describes the doubling of transistors in integrated circuits (IC) in computer hardware in a span of approximately two years (Intel Corporation, 2014). These shows processing speeds will not be a limiting factor for compression in future.

$$T \text{ (seconds)} = (E \div N) - (Z \div C) \tag{2.3}$$

Gzip compression algorithm has been adopted by web browsers (Constantin, 2011) and web servers (Microsoft, 2014) (Guillermo, 2011) as a way of compressing data in client-server communication model. Web browsers can decompress and render Gzip files compressed by web servers. This has improved bandwidth utilization and response time of files fetched to and fro the web server.

2.5.3 SERVER-SIDE TECHNIQUES

A server is a computer that receives and processes requests from client(s). Normally a server computer has high hardware specifications in order to process its client's requests efficiently. All the computing operations involved in server computer are said to exist in the Server-side. In this section we bring forth some of the techniques involved in the server-side operations in relation to SOAP which are: server-side caching technique and differential deserialization (DDS). These techniques are addressed as follows.

(i) Server-side Caching

Server-side caching is the storage of data on the server-side. Server-side caching improves response time (Seyyed et al., 2011) (Hou et al., 2010) (Matthieu et al., 2009). As discussed in client-side caching in section 2.5.1 (Client-side caching), server-side caching is slightly different as data is temporarily stored in serialized objects (Matthieu et al., 2009).

Hou et al. (2010) categorized cache in two methods: message body and template cache.

- Message body: This involves storing all SOAP message body calls. According to the client's request, the cache identifies each message body with two parameters: unique ID and Time To Live (TTL). If the requested XML message exists in cache and the TTL is valid, then the message is fetched from the cache and returned accordingly. Otherwise it is fetched in the server.
- Template cache: In this method, it is argued that in a service process, clients request the same elements but with different real time values. With this technique, the elements can form the template as the real time values are dynamically interchanged. This avoids reconstruction/destruction of the template message. Just like the message body technique functions, unique IDs are generated for message bodies and validated against when a client makes requests vis-à-vis its TTL in the template cache. This is managed by the template cache management module. Template cache structure is as shown in Figure 2.22.

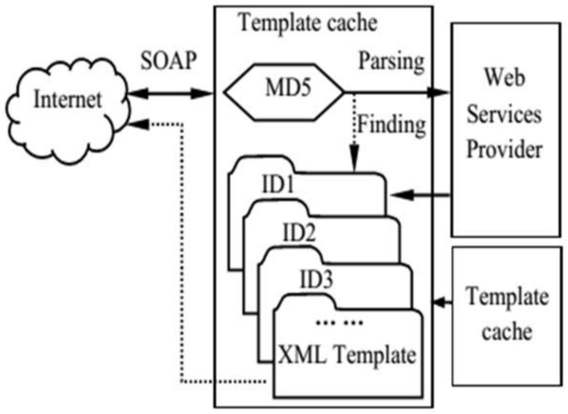


Figure 2.22: Structure of Template Cache

(Adopted from Hou et al. (2010))

Results of optimized message and template cache are as shown in Figure 2.23. Template caching is seen to have better performance than message body caching technique.

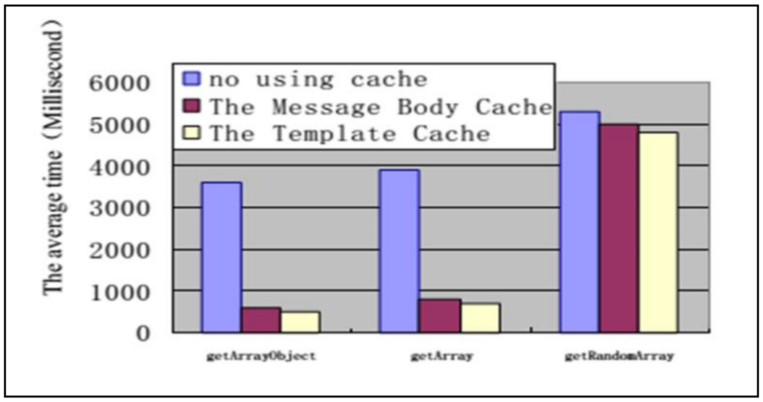
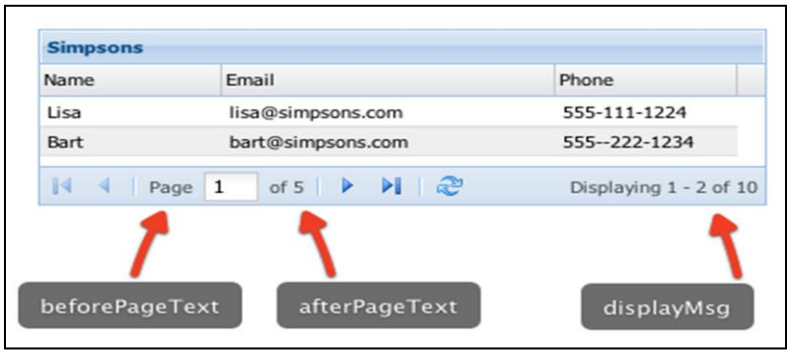


Figure 2.23: Test results for template and message body caching

(Adopted from Hou et al. (2010))

Server-side data chunking is one technique that is typically handled on the server-side. As the number of records to be loaded on the browser increases, the time required to load the data increases. Data chunking comes in as a very important technique for the response from the server to return a sustainable amount of data to the client. In Data chunking, the client specifies the range of data in the request; though this is handled programmatically. The server then composes the chunk and returns it via the response method. This improves performance of loading thousands of data, by loading chunks or bits (Chandra et al., 2012) (Sencha, 2013). Among other JavaScript libraries, Ext. JavaScript 4.1 (Sencha, 2013) has adopted this technique as paging which is quite essential in modeling controls e.g. grid as shown in Figure 2.24.

Similarly as discussed in section 2.5.1 (client-side caching), the challenge of keeping the cache up-to-date is also a major problem in server-side caching (Matthieu et al., 2009). Nevertheless, Matthieu et al. (2009) and Junichi et al. (2005) proposed the use of database-aided caching technique in the web server. Database caching comes with many advantages such as: modifications can be done easily, it has the ability to augment data with metadata, it eliminates the need to parse an entire XML structure which is computationally expensive, and an extra column can be used to store the aforementioned last modified timestamp value for each record easily. A suitable predicate value can be stored in replacement of last modified timestamp value to allow quick comparison between requests. Session management between multiple clients and proxy web server is a rich area of study that can be exploited further (Junichi et al., 2005) (Matthieu et al., 2009).


Simpsons

Name	Email	Phone
Lisa	lisa@simpsons.com	555-111-1224
Bart	bart@simpsons.com	555--222-1234

Figure 2.24: Data chunking/paging example in Ext. Js.

(Adopted from Sencha (2013))

(ii) Differential deserialization

Differential Deserialization (DDS) works in the server/receiver side. DDS technique has been supported in improving SOAP performance (Seyyed et al., 2011) (Tekli et al., 2011) (Nayef et al., 2004) (Behrouz et al., 2009) (Suzumura et al., 2005). DDS works best if similar messages are sent by different clients (Seyyed et al., 2011) (Nayef et al., 2004). DDS is somehow similar to Differential Serialization (DS). DDS and DS take advantage of a sequence of similar messages to avoid the expensive SOAP message de-serialization/serialization process respectively. Neither of them changes the SOAP protocol, the SOAP message nor SOAP message wire format. Both implementations remain independent and interoperable with other SOAP implementations (Nayef et al., 2004). Moreover, Nayef et al. (2005) noted that DDS is more promising implementation technique than DS because DS works if the same client sends a stream of similar messages whereas DDS avoids deserialization of similar messages sent by multiple clients. The speed of the server among other factors determines performance in DDS. Some of the differences between DDS and DS are captured in Table 2.3.

Table 2.3: Comparison between differential deserialization and differential serialization

Differential deserialization	**Differential serialization**
Works in the server-side	Works in the client-side
Deserialization process involves converting of SOAP XML-messages to application object	Serialization process involves conversion of in-memory data types to SOAP XML ASCII format
Uses parse checkpointing the state of the deserializer for incoming messages	Uses Data Update Tracking (DUT) Table to track whether a program has changed data items in new messages since they were last serialized SOAP messages in outgoing message

Deserialization is an expensive process that involves conversion of SOAP XML-messages to application object. Deserialization involves a series of undertakings which include fetching an appropriate deserializer from the type mapping registry, and constructing a Java object from an XML message. To be precise and relatively simple, the process of de-serializing an XML message into Java objects is as follows: (Suzumura et al., 2005)

- Open an XML document that represents the object.
- Reclusively de-serialize the object's members which are encoded as sub-elements after locating an appropriate deserializer from the type mapping system.
- Create a new instance of the Java type, initializing it with deserialized members.
- Return the new Java object.

These undertakings become more complex and expensive when the XML message becomes bigger and deeper. Deserialization can be improved by processing new regions of the XML messages and reuse of the constructed objects deserialized in the past. It even becomes more expensive when handling scientific data stored in arrays, floats, and doubles (Suzumura et al., 2005) (Behrouz et al., 2009) (Nayef et al., 2006). Objects are created before they are used and garbage collected after their use is over. The use of more objects affects the performance of garbage-cycling process (Suzumura et al., 2005).

Takase et al. (2005) noted that reuse of the entire object trees works for messages that have exactly the same structure. Suzumura et al. (2005) improved on the approach presented by Takase et al. (2005). Suzumura et al. (2005) noted that the fundamental characteristic of processed SOAP-based web services messages is that, their wire format structure has lots of similarity. By exploiting this weakness redundancy can be avoided by using a deserialization mechanism that reuses matching structures/objects from previously deserialized application's objects; deserialization for new regions is only performed on regions that will not have been processed before. Suzumura et al. (2005) obtained a 288% maximum performance gain by the use of this technique. However, in large messages on the wire that have repetitive elements like GoogleSearchLargeservice and in cluster-like environments, in such cases, reusing the entire object tree is not the optimal solution because repetition number might differ for each request and requires us to consider issues such as thread safeness and scalability respectively (Suzumura et al., 2005).

Nonetheless, DDS primary shortcomings in SOAP message exchange are processing of XML data/content and conversion of strings to in-memory data types; checkpointing is explored further (Nayef et al., 2006). DDS works by periodically checkpointing the state of SOAP deserializer, which reads and de-serializes incoming message portions, and computing checksum of these SOAP message portions. The checksum is compared against those of the corresponding message portion in the previous message. If the checksums match, the deserializer avoids redoing de-serializing (parsing and converting SOAP message) contents in that region. Essentially, the deserializer runs in two different modes: regular and fast mode. In regular mode, the deserializer reads and processes all the SOAP tags and message content as it creates checkpoints and corresponding message checksum along the way to the end of the SOAP message, whereas, in fast mode, the deserializer considers the sequence of checksum of each disjointed portions of the message and compares them against the sequence of checksums associated with the most recent received message.

The deserializer switches between regular and fast mode appropriately. Fast mode is switched on if the parser state is the same (checksum match) as the one that has been saved in a checkpoint. Regular mode is switched on when there is a checksum mismatch. This indicates a difference in the incoming message and the corresponding previous message. The deserializer switches from fast to regular mode where it reads and converts the message portion's content. Regular mode is actually the normal parsing without DDS optimization. In terms of performance, in fast mode, in the best scenario (when all the message portions are identical, though unrealistic), the normal cost of de-serializing is replaced by the cost of computing and comparing checksums which in general is significantly faster. In regular mode, the worst case scenario (when all the message portions are not identical), the DDS enabled deserializer runs much slower than a normal deserializer because it does the same work plus the added work of calculating checksums and creating parser checkpoints (Nayef et al., 2006). Further, Nayef et al. (2005) and Nayef et al. (2006) noted that creating many checkpoints can increase fast mode performance in terms of speed at the expense of checkpoint creation time, check point memory utilization, and checksum calculation and comparison time. Actually checkpointing is memory intensive.

Due to the relatively high memory requirements experienced in checkpointing, Nayef et al. (2005), Nayef et al. (2006) introduces a new technique for storing only the difference between successive parser states for messages. This technique is called Differential Checkpointing (DCP). DCP involves only the differences between the consecutive checkpoints as opposed to storing the entire parse states for each checkpoint. DCP optimizes DDS by improving its speed and reducing memory requirements. Despite the fact that DCP reduced memory requirements, it still required significant processing overheads. Moreover, DDS primary shortcomings in its implementation are generating, storing, and using parse checkpoints. Nayef et al. (2006) introduced Lightweight Checkpointing (LCP); the checkpointing approach significantly reduced the cost of both DCP and DDS techniques. LCP checkpoints contain very little state information (fewer bytes) created at predefined points within the structure of the message. Each lightweight checkpoint in LCP has a reference to a base checkpoint that contains state information it shares with other lightweight checkpoints. In LCP, creation of checkpoints is much faster than regular checkpoints, hence requires much less memory and requires less processing overheads. LCP takes only 10% of memory that DCP requires and 3% of the memory original checkpointing algorithm required. For processing time, deserialization with LCP was approximately 36% better than DCP and approximately 52% better than Full Checkpointing (FCP), on average, when approximately half of the message is not changed from the previous message. In FCP, the full parser state is stored with each checkpoint.

Moreover, Behrouz et al. (2009) suggested a Serialization Enhancement Middleware (SEM) Technique that utilizes a combination of DS and DDS techniques to improve response time. SEM is an implementation that runs on the middleware to run on top of any web server. SEM acts as the primary module and takes advantage of similar SOAP requests in a web server.

Similarly, SEM avoids redundant serialization stage of SOAP response for request which have completely the same parameters. SEM maintains a trie of incoming parameters for current requests thus processing and serialization of response of same requests is done only once.

2.6 RESEARCH GAPS

In SOA, web services provide a comprehensive solution for representing, discovering and invoking services in distributed systems. At the core of the web services, lie various XML-based standards including SOAP. SOAP is a protocol that ensures web services extensibility, robustness and interoperability between heterogeneous systems. From previous researchers' literature addressed in section 2.5, it is evident that SOAP's dependence on XML has basically two major performance-related drawbacks:

(i) XML structure is verbose which results to high network traffic and poor network utilization

(ii) XML parsing and processing causes a high computational burden leading to high latency

2.7 DEVELOPED TECHNIQUE TO IMPROVE SOAP PERFORMANCE

Several researchers in section 2.5 have made contributions of how to address the research gaps indicated in section 2.6. The scope of this research addressed the verbosity of SOAP messages which results to huge messages thus poor bandwidth utilization. In relation to this research's literature survey of techniques of improving SOAP performance covered in section 2.5, this research developed an aggregation of the following techniques: client-side caching, document-literal, simple database queries on the server-side and Gzip compression techniques.

These techniques are implemented at the presentation layer of the OSI (Open Systems Interconnection) model. ASP.NET web services were used in developing this research's web services. This research adopts and modifies the architecture proposed by Seyyed et al. (2011) shown in Figure 2.19 in aggregating these techniques. Figure 2.25 shows this research's hybrid architecture to improve SOAP performance.

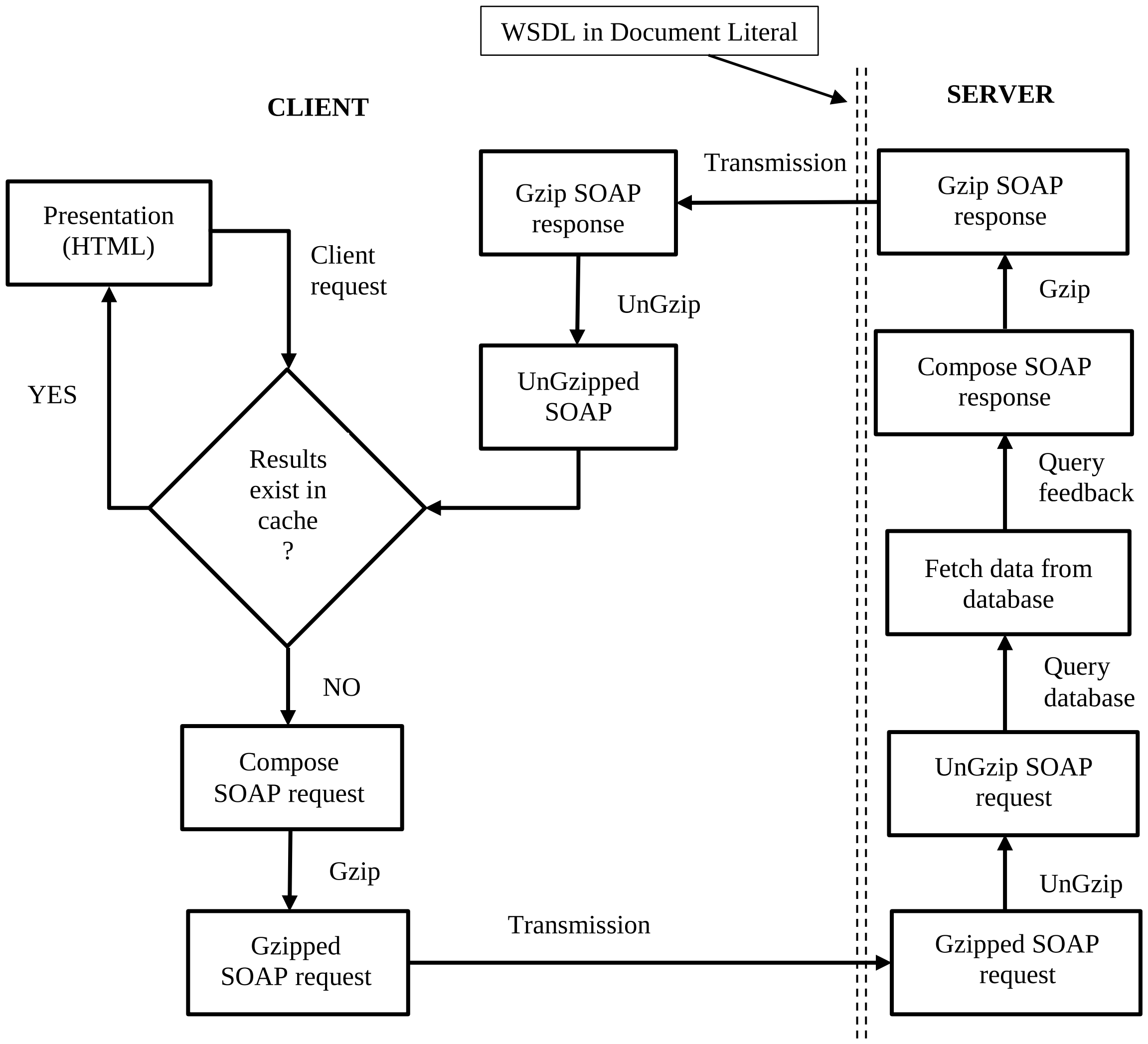


Figure 2.25: The hybrid architecture for improving SOAP performance

2.8 SUMMARY

This chapter has several sections that take us through inception stages of SOAP in the SOA to its performance challenges, contributions from other researchers then this research's developed SOAP performance hybrid architecture. We have seen that SOA can be implemented in web services, Java RPC, CORBA, COM etc. A web services is an abstract interface whose services can be invoked programmatically using SOAP or REST techniques. Web services are interoperable in heterogeneous systems. SOAP is more secure than REST. Nevertheless, several researchers have shown that SOAP's dependence on XML is messaging is its major undoing in performance.

Several techniques have been proposed and implemented in improving SOAP performance. This research categorized these techniques thematically as client-side, communication channel and server-side techniques. The some of the techniques that fall under these thematic areas have been discussed. From this literature, we identified major research gaps in SOAP performance. Out of the weaknesses identified in models proposed by other researchers, we proposed a hybrid architecture that aggregates and blends client-side caching, document-literal, simple database queries on the server-side and Gzip compression techniques. Chapter Three take us through development of the software that realized this hybrid architecture.

CHAPTER THREE

SOFTWARE DEVELOPMENT

3.1 SOFTWARE DEVELOPMENT PROCESS

This research adopted waterfall model of software development. The steps undertaken in this research's software development were as follows (Sommerville, 2011).

3.1.1 REQUIREMENTS ELICITATION AND ANALYSIS

This research followed requirements elicitation guidelines which are summarized in the following steps (Pressman, 2010):

- Assessing business and technical feasibility for the proposed architecture.
- Defining the technical environment into which the software was running e.g. computer speed, RAM, operating system, communications, and software platform.
- Identifying the domain constraints that limit the functionality or performance of the software to be built. That is, the characteristics of the business environment specific to the application domain.
- Use of requirements elicitation methods including; interviews and focus groups.
- Soliciting participation from people so that requirements are defined from different points of view e.g. brainstorming.
- Reading secondary data sources. This enabled this research to establish how other researchers modeled their systems and the constraints they put in place.

The requirements elicitation process and analysis processes led to establishing software specifications. These specifications govern the functional boundaries of the system i.e.

software's services, constraints, and goals (Sommerville, 2011). This research identified the following system requirements:

(i) Ability to handle specified size of data and render appropriately: This research measured large file sizes of up to 7300KB and a small file size of up to 200KB. The software was to have the capability of handling any file size in that range.

(ii) Platforms operability: This system executes in Windows 2000 Server and Windows XP clients. The software was to run on these environments. The software developer was to use .NET 2.0 framework while compiling this software because it is supported in Windows 2000 Server and Windows XP. The client software to be compiled in .NET framework 3.0; it is supported in Windows XP.

(iii) Local host web hosting and database: The SOAP web service application was hosted in Windows 2000 Server IIS while the web clients were hosted in Windows XP IIS. The database server was operable in the computer hosting Windows 2000 Server.

(iv) LAN set up: Server computer and the client computers were linked with a switch/ hub in a network cable. All cables to and fro the client computers and the switch were one meter long. Client computers were to call services from server computer.

(v) Caching of data on the client-side: The software was to cache data on the client-side. This was done using Ext. Js which is a JavaScript library. This data was fetched from the .aspx ASP.NET back end code file.

(vi) SOAP request and responses: The software was to send SOAP requests to the server. The server was to compose SOAP responses accordingly. The web service was to pick parameters describing the file size/data required to be fetched from server database. The server processed the request and composed the appropriate file size and sent it back to the client computer.

(vii) SOAP web service calls: The software made web service calls while the client computers ASP.NET .aspx back end code file to the .asmx SOAP web service.

(viii) Implementing simple database queries: The database was behind the web service. The web services linked with the database and fetch the appropriate sizes of data. Simple database queries were executed. The database was MySQL. The database was directly accessible by server computers and not the client computer.

(ix) Implementation of document-literal technique in WSDL: The software had document-literal implemented in describing WSDL.

(x) Implement Gzip algorithm in compressing responses: The web server was able to compress responses. Uncompressed responses were equally required.

3.1.2 SYSTEM AND SOFTWARE DESIGN

System and software design involves identifying and describing the fundamental software system abstractions and their relationships (Sommerville, 2011). This research described the model and specifications of the system software using UML (Unified Modeling Language). UML models can be categorized in three: static, functional/behavioral and dynamic modeling. This research designed at least one model: class diagrams (static model), activity diagram (functional), and sequence diagram (dynamic). Besides UML diagrams, Figure 3.4 shows the general client-server communication model.

(i) Class diagram – static model

A class diagram describes the structure of a system by showing the system's classes, their attributes, operations, and the relationships among objects. Figure 3.1 shows class diagrams of three classes: Data, Data_DL and Connectivty. The class Data had two properties (_Text_ID and _Text_Data), two get set methods (Text_ID and Text_Data), and one constructor method – Data(). The class Data_DL implemented the logic. It had two methods Data_RetrieveServer() which fetched data from the database and TypeCast() that instantiates a class Data object and sets its properties (Data_ID and Data_Text) with values from the database.

Nevetheless, class Data_DL inherits from the class Connectivity that opens connectivity to the database using its default constructor and closes connectivity to the database using its default destructor. Class connectivity has inheritable properties (Comm, Conn and myReader). Comm property is the MySqlCommand. Comm is the MySqlConnection while myReader is MySqlDataReader.

(ii) Activity diagram – functional model

An activity diagram is a graphical representation of workflows of stepwise activities and actions with support for choice, iteration and concurrency. The activity diagram in Figure 3.2 shows steps and flow of processes followed when a client requested for data from the server. If the requested data exited in the server, the web service composed a response and sent it to the client.

(iii) Sequence diagram – dynamic model

A sequence diagram is a type of interaction diagram. It models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case. Figure 3.3 show an activity diagram of this research's client-server communication model. The computer user used the homepage of the client web application. The client web application, from the backend code sent a request to the web service requesting for a certain size of data. The web service opened connectivity to the database then run an SQL statement. The database then returned data to the web service. The web server, which hosted the web service, composed a response and sent the data back to the client application. The client form, application back end code, received the data and forwards it to the front end interface. The front end interface was purely in JavaScript; it renders the data.

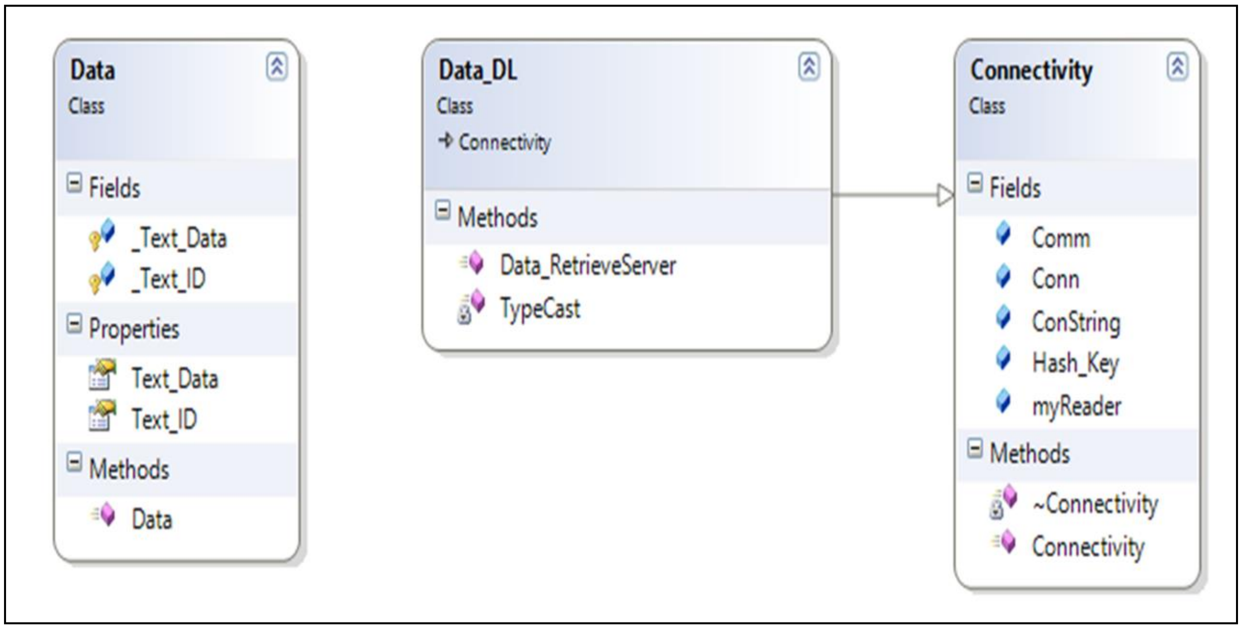


Figure 3.1: Class diagrams showing three classes: Data, Data_DL and Connectivty

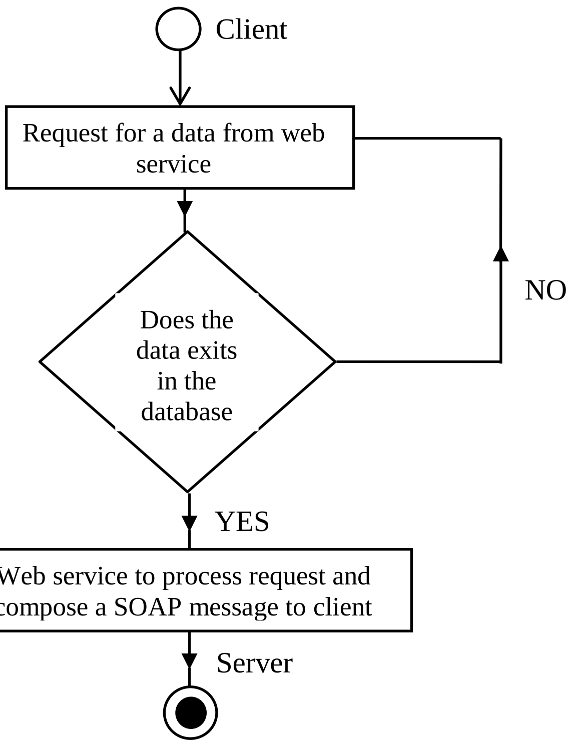

Client
Request for a data from web service
Does the data exits in the database
NO
YES
Web service to process request and compose a SOAP message to client
Server

Figure 3.2: An activity diagram for client - server request response

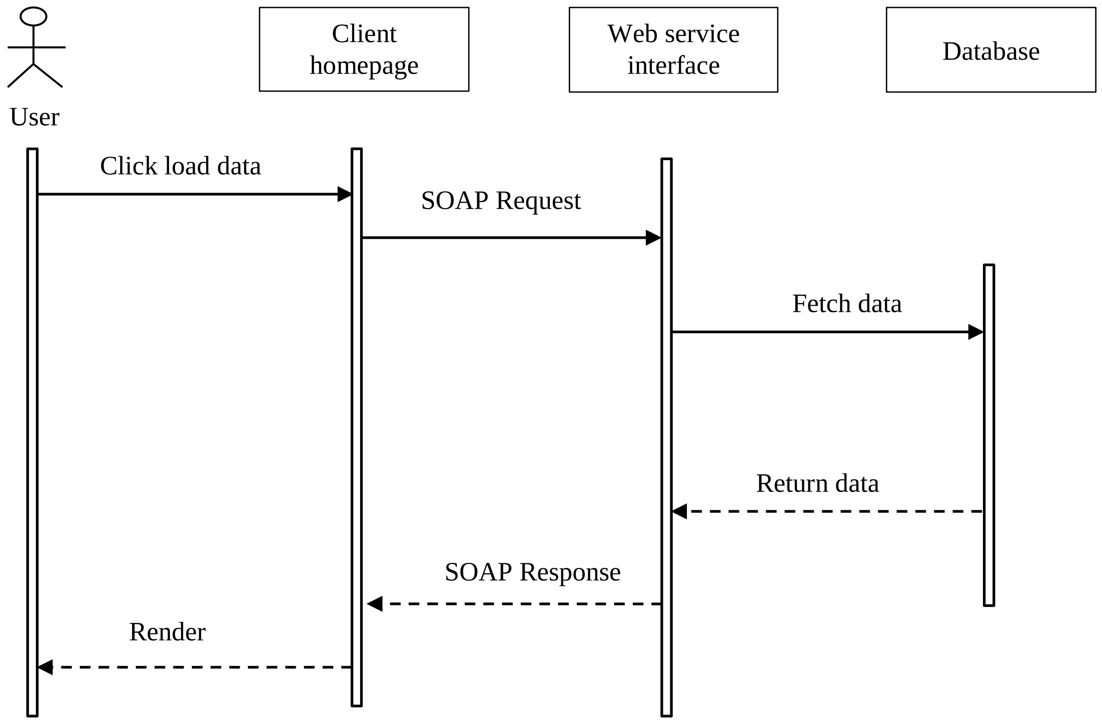


Figure 3.3: A sequence diagram for client - server communication

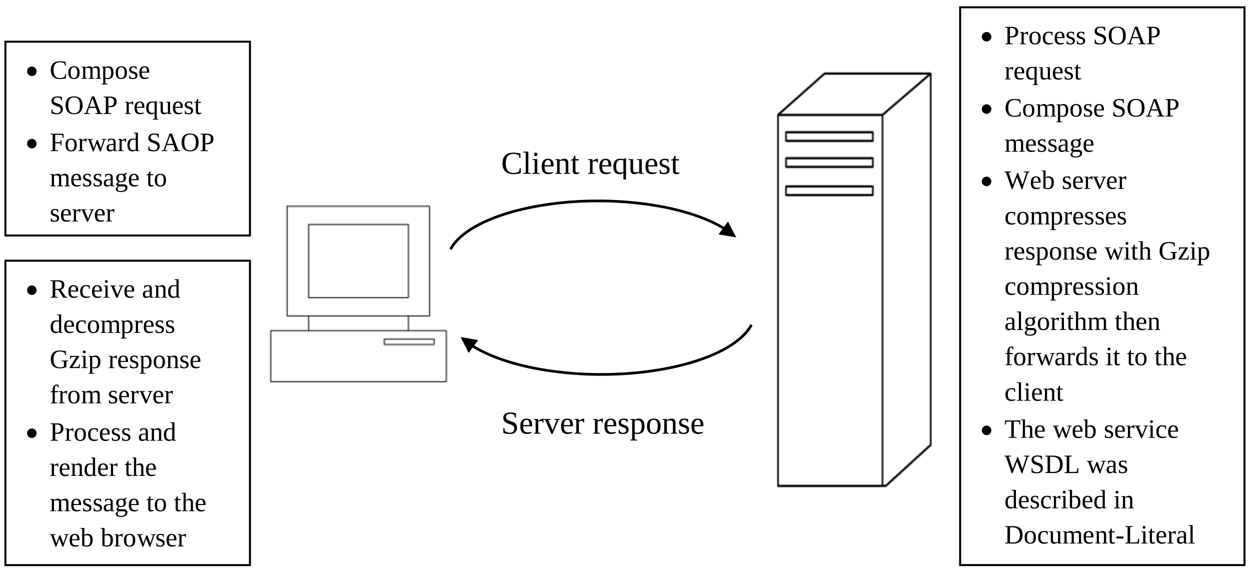

• Compose SOAP request
• Forward SAOP message to server
• Receive and decompress Gzip response from server
• Process and render the message to the web browser
Client request
Server response
• Process SOAP request
• Compose SOAP message
• Web server compresses response with Gzip compression algorithm then forwards it to the client
• The web service WSDL was described in Document-Literal

Figure 3.4: Client - server communication

3.1.3 IMPLEMENTATION AND UNIT TESTING

In this stage, the software design was realized as a set of programs or program units. Microsoft Visual Studio 2010 was selected as the appropriate IDE (Integrated Development Environment) for software development. The software had three major components to be developed: Client-side component, Server-side component and the database component.

- The client-side was built in Ext. Js. which is a JavaScript library, ASP.NET and C# as the backend programming language.
- The server-side was built using .asmx web service and C# as the back end language.
- The database component was built on MySQL Server open source database.

Basic coding standard were maintained e.g. commenting program code, file organization etc. Unit testing involves verifying that each unit meets its specification. Component testing was to verify that each component meets its specification. A components is a collection of classes, interfaces etc. In this research, each component was tested to ensure that they work correctly. The client application interface is show in Figure 3.5 and web service in Figure 3.6

Notwithstanding, Shariq et al. (2012) did a study of web services testing tools: soapUI (SmartBear, 2013), JMeter (JMeter, 2013) and Storm (Storm, 2013). Testing was in terms of their architecture, features, interoperability, software requirements, and usability. Moreover, they did a comparison of their throughput, response time and usability; only JMeter and soapUI support testing of throughput. Nonetheless, soapUI outperformed JMeter and Storm thus can be regarded as fastest tool in terms of response time, JMeter had better throughput than soapUI, and Storm had a very simple and easy to use interface. It was observed that response time values taken at 6:00 AM are most optimal (Shariq et al., 2012).

Apache Bench can be used to test various metrics among them: throughput and response time (Apache Foundation, 2013). Web services performance testing tools is a rich area of study that can exploited. Nevertheless, there are many vendors in the internet who have come up with tools that can test web services. Some of these tools include: Fiddler Web Proxy (Terelik, 2013), NetMon (Microsoft, 2013), Wire Shark (Wireshark), and NeoLoad (NEOTYS, 2013). Software testing in complex systems can be very involving. Software performance profiling can be very essential in determining a software’s performance in terms of memory utilization, execution time etc. (Mostafa, 2008) (Terrence, 2013).

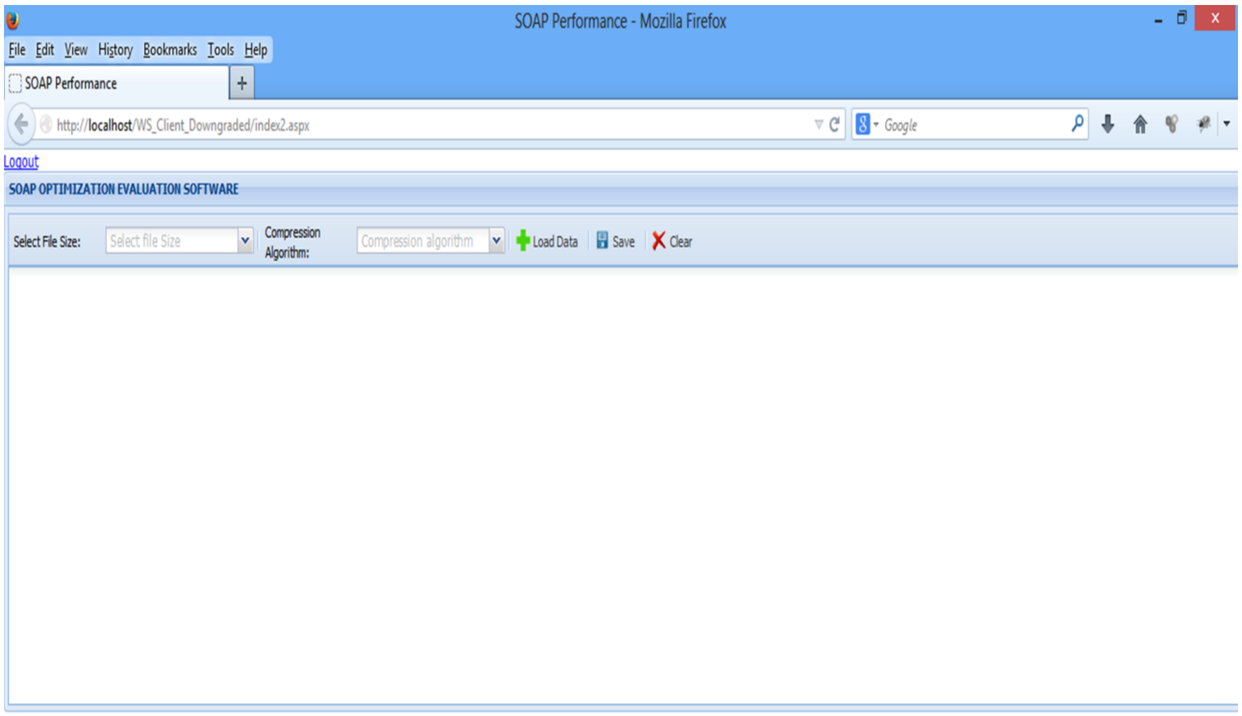
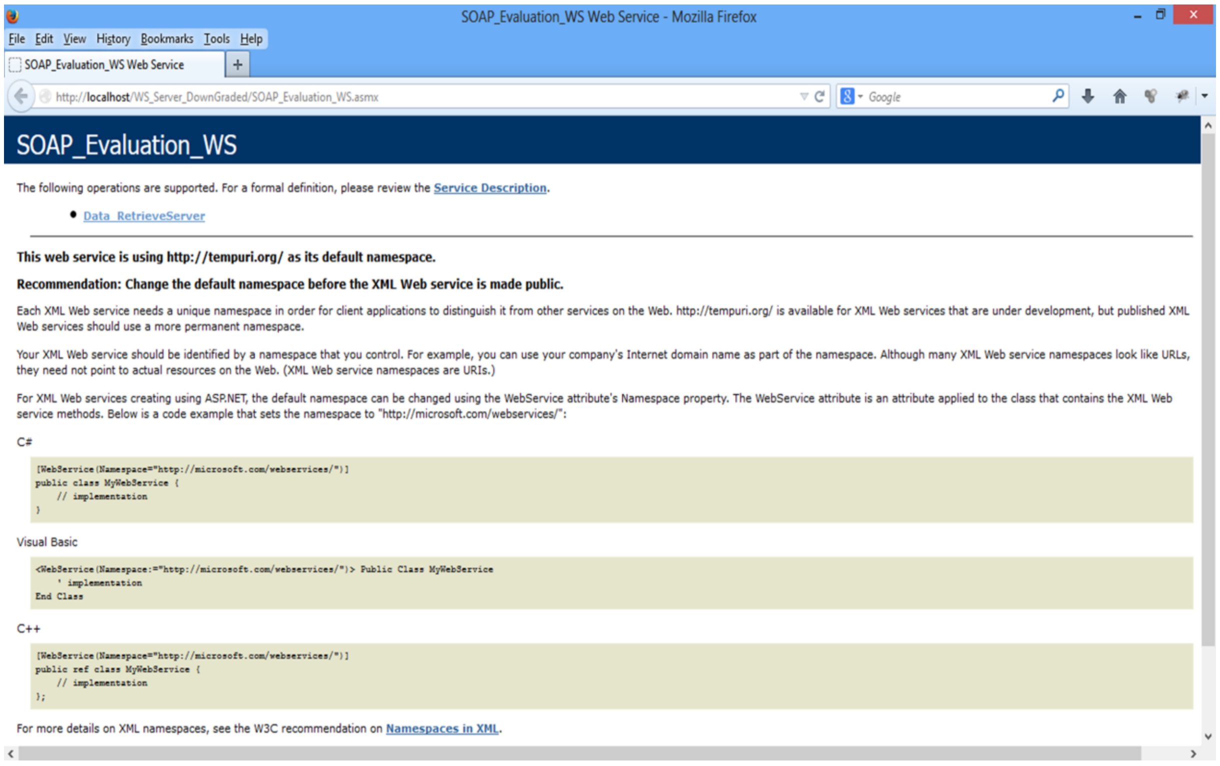

SOAP Performance - Mozilla Firefox
File Edit View History Bookmarks Tools Help
SOAP Performance
http://localhost/WS_Client_Downgraded/index2.aspx
Google
Logout
SOAP OPTIMIZATION EVALUATION SOFTWARE
Select File Size:
Select file Size
Compression Algorithm:
Compression algorithm
Load Data
Save
Clear

Figure 3.5: Software client application interface developed in ASP.Net and JavaScript


SOAP_Evaluation_WS Web Service - Mozilla Firefox
File Edit View History Bookmarks Tools Help
SOAP_Evaluation_WS Web Service
http://localhost/WS_Server_DownGraded/SOAP_Evaluation_WS.asmx
Google

SOAP_Evaluation_WS

The following operations are supported. For a formal definition, please review the Service Description.

- Data_RetrieveServer

This web service is using http://tempuri.org/ as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other services on the Web. http://tempuri.org/ is available for XML Web services that are under development, but published XML Web services should use a more permanent namespace.

Your XML Web service should be identified by a namespace that you control. For example, you can use your company's Internet domain name as part of the namespace. Although many XML Web service namespaces look like URLs, they need not point to actual resources on the Web. (XML Web service namespaces are URIs.)

For XML Web services creating using ASP.NET, the default namespace can be changed using the WebService attribute's Namespace property. The WebService attribute is an attribute applied to the class that contains the XML Web service methods. Below is a code example that sets the namespace to "http://microsoft.com/webservices/":

C#

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public class MyWebService {
    // implementation
}
```

Visual Basic

```
<WebService(Namespace:="http://microsoft.com/webservices/")> Public Class MyWebService
    ' implementation
End Class
```

C++

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public ref class MyWebService {
    // implementation
};
```

For more details on XML namespaces, see the W3C recommendation on Namespaces in XML.

Figure 3.6: Web service application interface showing a public method, “Data_RetrieveServer”. Methods can be invoked programatically using SOAP

3.1.4 COMPONENT INTEGRATION AND SYSTEM TESTING

Component Integration involved bringing together all the fully developed and tested components. In this research we integrated together the client, server and the database components. System testing was to verify that all the integrated components meet the overall software requirements (Sommerville, 2011).

3.1.5 OPERATION AND MAINTENANCE

This involved the system being installed and put into practical use in the computer lab where we ran the experiments. Maintenance involved correcting errors which were not discovered in the earlier stages of system development.

CHAPTER FOUR

EXPERIMENTS, RESULTS AND DISCUSSION

4.1 EXPERIMENT SETUP

We set up an experimental design where we manipulated the file size of responses from the web server as we set constant other experiment environmental conditions. The experimental environment was controlled to so that we attain accurate and valid experimental results. The experimental environments were as follows:

- Windows 2000 Server OS installed in one server computer and Windows XP installed in three client computers. The client computers are named as COMP A, COMP B, and COMP C.
- Network bandwidth set to 10mbps; simulates a disadvantage/poor network bandwidth.
- All computers had 1GB RAM and 3.2GHz.
- Client and server were interconnected with a switch.
- Web services were developed in ASP.NET. ASP.NET web services were compressed and uncompressed. Leading to the hybrid architecture be hybrid compressed and hybrid uncompressed respectively. Compression was done using Gzip algorithm.
- Web applications were hosted in Microsoft Internet Information Services (IIS).

We manipulated file size retrieved from the web server by fetching specific quantified data from the database. Extraneous variables include network traffic and background processes running in the computers. In a bid to control extraneous variables we ensured that:

- All computers were formatted.
- Relevant programs to this research were installed e.g. .NET SDK’s in all the computers.

- Browser caches, cookies and offline stored data were deleted before running experiment.
- We waited the response to load, recorded Fiddler Web Proxy results then ran the next experiment.
- Microsoft Windows native applications were to be used e.g. internet explorer and IIS.

Nevertheless, in section 2.4.4 it was noted that SOAP rides on HTTP. Fiddler software monitors HTTP traffic and gives actual performance statistics which this research is interested in. That aside, NetBalance Software was used to limit bandwidth to 10mbps. A bandwidth of 10mbps exposes the application to poor and strenuous bandwidth which this research is interested in; disadvantaged network. This research used a switch to interconnect the server and client computers. Experimental raw data was collected as outlined in section 4.2.

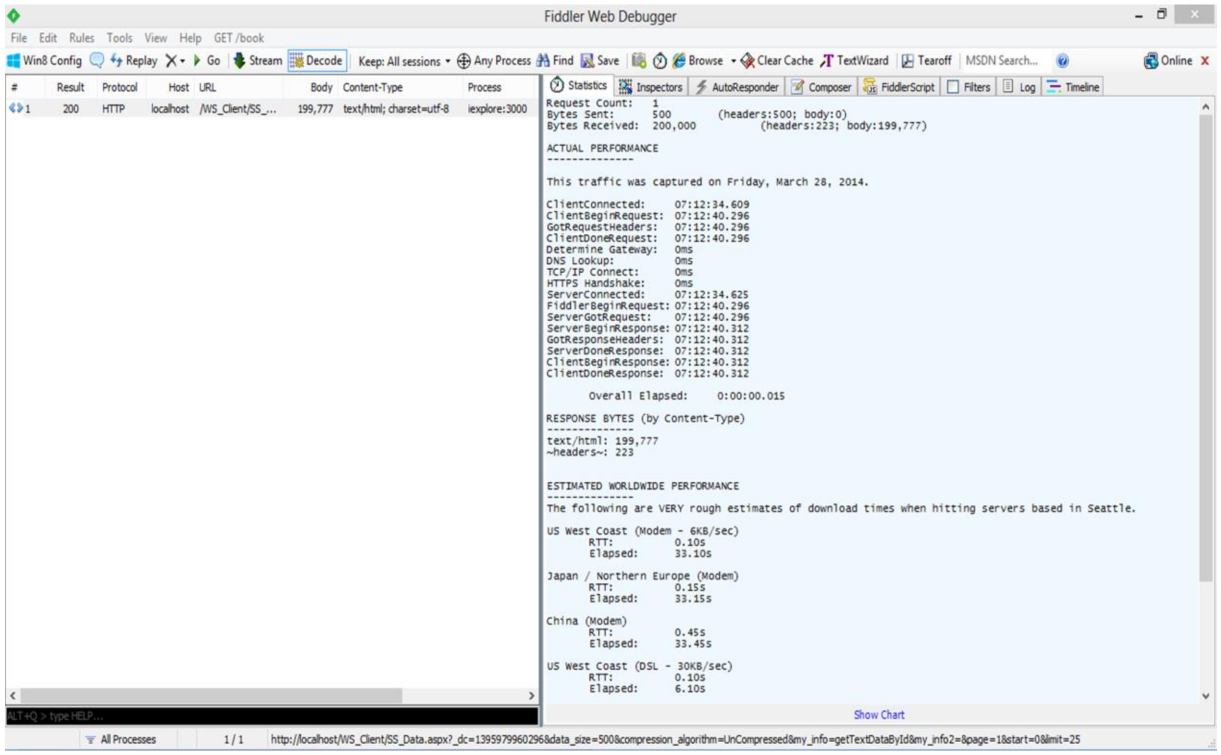
4.2 EXPERIMENT RAW DATA

Table 4.1, Table 4.2, and Table 4.3 show raw hybrid compressed HTTP SOAP traffic collected from computer A, B, and C respectively. Table A-1, Table A-2, and Table A-3 show raw hybrid architecture uncompressed HTTP SOAP traffic collected from computer A, B, and C respectively. Fiddler Web Proxy software was used in collecting data. A sample of Fiddler Web Proxy software interface showing actual performance is shown in Figure 4.1 and Figure 4.2.

A brief description of the data representation Table A-1, Table A-2, and Table A-3 is shown below (Fiddler, 2014):

- ClientConnected - Exact time that the client browser made a TCP/IP connection to Fiddler.
- ClientBeginRequest - Time at which this HTTP request began. May be much later than ClientConnected due to client connection reuse.

- ClientDoneRequest - Exact time that the client browser finished sending the HTTP request to Fiddler.
- DNSTime - Milliseconds Fiddler spent in DNS looking up the server's IP address.
- GatewayDeterminationTime - Milliseconds Fiddler spent determining the upstream gateway proxy to use (e.g. processing autoproxy script). Mutually exclusive to DNSTime.
- TCPConnectTime - milliseconds Fiddler spent TCP/IP connecting to that server's IP address.
- HTTPSHandshakeTime - Amount of time spent in HTTPS handshake
- ServerConnected - Time at which this connection to the server was made. May be much earlier than ClientConnected due to server connection reuse.
- FiddlerBeginRequest - The time at which Fiddler began sending the HTTP request to the server.
- ServerGotRequest - Exact time that Fiddler finished (re)sending the HTTP request to the server.
- ServerBeginResponse - Exact time that Fiddler got the first bytes of the server's HTTP response.
- ServerDoneResponse - Exact time that Fiddler got the last bytes of the server's HTT response.
- ClientBeginResponse - Exact time that Fiddler began transmitting the HTTP response to the client browser.
- ClientDoneResponse- Exact time that Fiddler finished transmitting the HTTP response to the client browser.

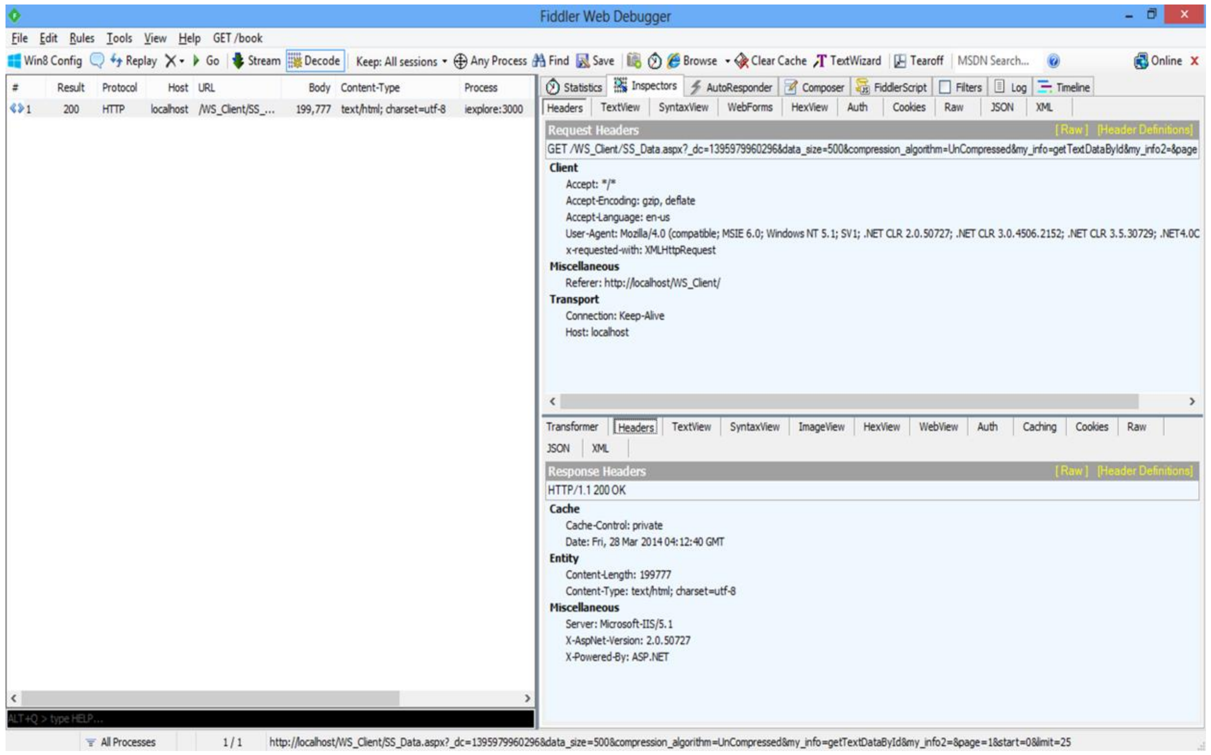


```
#    Result  Protocol  Host       URL              Body     Content-Type              Process
1    200     HTTP      localhost  /WS_Client/SS_...  199,777  text/html; charset=utf-8  iexplore:3000
```

```
Request Count:   1
Bytes Sent:      500       (headers:500; body:0)
Bytes Received:  200,000          (headers:223; body:199,777)

ACTUAL PERFORMANCE
--------------

This traffic was captured on Friday, March 28, 2014.

ClientConnected:     07:12:34.609
ClientBeginRequest:  07:12:40.296
GotRequestHeaders:   07:12:40.296
ClientDoneRequest:   07:12:40.296
Determine Gateway:   0ms
DNS Lookup:          0ms
TCP/IP Connect:      0ms
HTTPS Handshake:     0ms
ServerConnected:     07:12:34.625
FiddlerBeginRequest: 07:12:40.296
ServerGotRequest:    07:12:40.296
ServerBeginResponse: 07:12:40.312
GotResponseHeaders:  07:12:40.312
ServerDoneResponse:  07:12:40.312
ClientBeginResponse: 07:12:40.312
ClientDoneResponse:  07:12:40.312

       Overall Elapsed:     0:00:00.015

RESPONSE BYTES (by Content-Type)
--------------
text/html: 199,777
~headers~: 223


ESTIMATED WORLDWIDE PERFORMANCE
--------------
The following are VERY rough estimates of download times when hitting servers based in Seattle.

US West Coast (Modem - 6KB/sec)
       RTT:          0.10s
       Elapsed:      33.10s

Japan / Northern Europe (Modem)
       RTT:          0.15s
       Elapsed:      33.15s

China (Modem)
       RTT:          0.45s
       Elapsed:      33.45s

US West Coast (DSL - 30KB/sec)
       RTT:          0.10s
       Elapsed:      6.10s
```

```
1 / 1    http://localhost/WS_Client/SS_Data.aspx?_dc=1395979960296&data_size=500&compression_algorithm=UnCompressed&my_info=getTextDataById&my_info2=&page=1&start=0&limit=25
```

Figure 4.1: Snapshot of a sample of Fiddler Web Proxy interface showing actual performance



```
Request Headers
GET /WS_Client/SS_Data.aspx?_dc=1395979960296&data_size=500&compression_algorithm=UnCompressed&my_info=getTextDataById&my_info2=&page
Client
    Accept: */*
    Accept-Encoding: gzip, deflate
    Accept-Language: en-us
    User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C
    x-requested-with: XMLHttpRequest
Miscellaneous
    Referer: http://localhost/WS_Client/
Transport
    Connection: Keep-Alive
    Host: localhost

Response Headers
HTTP/1.1 200 OK
Cache
    Cache-Control: private
    Date: Fri, 28 Mar 2014 04:12:40 GMT
Entity
    Content-Length: 199777
    Content-Type: text/html; charset=utf-8
Miscellaneous
    Server: Microsoft-IIS/5.1
    X-AspNet-Version: 2.0.50727
    X-Powered-By: ASP.NET
```

Figure 4.2: Snapshot of a sample of Fiddler Web Proxy interface showing request and response headers

Table 4.1: COMP A - raw Gzip compressed HTTP SOAP traffic

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500**	**501**	**501**	**501**	**501**
Sent: Body (KB)	**0**	**0**	**0**	**0**	**0**
Received: Header (KB)	**246**	**247**	**247**	**248**	**248**
Received: Body (KB)	**62,722**	**124,144**	**802,382**	**1,857,706**	**2,466,262**
ClientConnected: (Time)	**19:08:19.109**	**19:08:19.109**	**19:09:46.937**	**19:09:46.937**	**19:09:46.937**
ClientBeginRequest: (Time)	**19:08:42.875**	**19:09:05.421**	**19:09:46.937**	**19:10:10.359**	**19:10:38.203**
GotRequestHeaders: (Time)	**19:08:42.875**	**19:09:05.421**	**19:09:46.937**	**19:10:10.359**	**19:10:38.203**
ClientDoneRequest: (Time)	**19:08:42.875**	**19:09:05.421**	**19:09:46.937**	**19:10:10.359**	**19:10:38.203**
Determine Gateway: (Time)	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
DNS Lookup:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
TCP/IP Connect:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
HTTPS Handshake:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
Time ServerConnected: (Time)	**19:08:42.875**	**19:08:42.875**	**19:08:42.875**	**19:08:42.875**	**19:08:42.875**
FiddlerBeginRequest: (Time)	**19:08:42.875**	**19:09:05.421**	**19:09:46.937**	**19:10:10.359**	**19:10:38.203**
ServerGotRequest: (Time)	**19:08:42.875**	**19:09:05.421**	**19:09:46.937**	**19:10:10.359**	**19:10:38.203**
ServerBeginResponse: (Time)	**19:08:42.921**	**19:09:05.500**	**19:09:47.375**	**19:10:11.328**	**19:10:39.453**
GotResponseHeaders: (Time)	**19:08:42.921**	**19:09:05.500**	**19:09:47.375**	**19:10:11.328**	**19:10:39.453**
ServerDoneResponse: (Time)	**19:08:42.921**	**19:09:05.500**	**19:09:47.375**	**19:10:11.343**	**19:10:39.484**
ClientBeginResponse: (Time)	**19:08:42.921**	**19:09:05.500**	**19:09:47.390**	**19:10:11.343**	**19:10:39.484**
ClientDoneResponse: (Time)	**19:08:42.921**	**19:09:05.500**	**19:09:47.390**	**19:10:11.343**	**19:10:39.484**

Table 4.2: COMP B - raw Gzip compressed HTTP SOAP traffic

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body (KB)	0	0	0	0	0
Received: Header (KB)	246	247	247	248	248
Received: Body (KB)	62,722	124,144	802,382	1,857,706	2,466,262
ClientConnected: (Time)	19:12:19.140	19:12:19.140	19:12:19.140	19:14:08.625	19:14:08.625
ClientBeginRequest: (Time)	19:12:45.046	19:13:09.390	19:13:25.625	19:14:08.625	19:14:33.468
GotRequestHeaders: (Time)	19:12:45.046	19:13:09.390	19:13:25.625	19:14:08.625	19:14:33.468
ClientDoneRequest: (Time)	19:12:45.046	19:13:09.390	19:13:25.625	19:14:08.625	19:14:33.468
Determine Gateway: (Time)	0ms	0ms	0ms	0ms	0ms
DNS Lookup:	0ms		0ms	0ms	0ms
TCP/IP Connect:	0ms	0ms	0ms	0ms	0ms
HTTPS Handshake:	0ms	0ms	0ms	0ms	0ms
Time ServerConnected: (Time)	19:12:19.156	19:12:19.156	19:12:19.156	19:12:19.156	19:12:19.156
FiddlerBeginRequest: (Time)	19:12:45.046	19:13:09.390	19:13:25.625	19:14:08.625	19:14:33.468
ServerGotRequest: (Time)	19:12:45.046	19:13:09.390	19:13:25.625	19:14:08.625	19:14:33.468
ServerBeginResponse: (Time)	19:12:45.078	19:13:09.484	19:13:26.031	19:14:09.609	19:14:34.734
GotResponseHeaders: (Time)	19:12:45.078	19:13:09.484	19:13:26.031	19:14:09.609	19:14:34.734
ServerDoneResponse: (Time)	19:12:45.078	19:13:09.484	19:13:26.031	19:14:09.625	19:14:34.750
ClientBeginResponse: (Time)	19:12:45.078	19:13:09.484	19:13:26.046	19:14:09.625	19:14:34.765
ClientDoneResponse: (Time)	19:12:45.078	19:13:09.484	19:13:26.046	19:14:09.625	19:14:34.765

Table 4.3: COMP C - raw Gzip compressed HTTP SOAP traffic

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body (KB)	0	0	0	0	0
Received: Header (KB)	246	247	247	248	248
Received: Body (KB)	62,722	124,144	802,382	1,857,706	2,466,262
ClientConnected: (Time)	19:14:30.015	19:14:30.015	20:18:12.531	19:14:30.015	19:14:30.015
ClientBeginRequest: (Time)	19:14:35.718	19:14:57.250	20:18:12.531	19:15:48.593	19:16:28.500
GotRequestHeaders: (Time)	19:14:35.718	19:14:57.250	20:18:12.531	19:15:48.593	19:16:28.500
ClientDoneRequest: (Time)	19:14:35.718	19:14:57.250	20:18:12.531	19:15:48.593	19:16:28.500
Determine Gateway: (Time)	0ms	0ms	0ms	0ms	0ms
DNS Lookup:	0ms	0ms	0ms	0ms	0ms
TCP/IP Connect:	0ms	0ms	0ms	0ms	0ms
HTTPS Handshake:	0ms	0ms	0ms	0ms	0ms
Time ServerConnected: (Time)	19:14:30.015	19:14:30.015	20:17:36.515	19:14:30.015	19:14:30.015
FiddlerBeginRequest: (Time)	19:14:35.718	19:14:57.250	20:18:12.531	19:15:48.593	19:16:28.500
ServerGotRequest: (Time)	19:14:35.718	19:14:57.250	20:18:12.531	19:15:48.593	19:16:28.500
ServerBeginResponse: (Time)	19:14:35.765	19:14:57.312	20:18:12.968	19:15:49.562	19:16:29.781
GotResponseHeaders: (Time)	19:14:35.765	19:14:57.328	20:18:12.968	19:15:49.562	19:16:29.781
ServerDoneResponse: (Time)	19:14:35.765	19:14:57.328	20:18:12.984	19:15:49.578	19:16:29.796
ClientBeginResponse: (Time)	19:14:35.765	19:14:57.328	20:18:12.984	19:15:49.578	19:16:29.796
ClientDoneResponse: (Time)	19:14:35.765	19:14:57.328	20:18:12.984	19:15:49.578	19:16:29.796

4.3 EXPERIMENT PRE-PROCESED DATA

In this section we refine the raw data collected as outlined in section 4.2. We then calculate the time since the client got connected to the server to send requests, to the time the client was done working on the response after the client received the response from the server. This time period has several check points e.g. ClientBeginRequest, GotRequestHeaders etc. as indicated in Table 4.4, Table 4.5, Table 4.6, Table B-1, Table B-2, Table B-3, Table 4.7, and Table 4.8.

Table 4.4: COMP - A Gzip compressed hybrid architecture pre-processed results

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body(KB)	0	0	0	0	0
Total Request Header (KB)	500	501	501	501	501
Received: Header (KB)	246	247	247	248	248
Received: Body (KB)	62,722	124,144	802,382	1,857,706	2,466,262
Total Response Header (KB)	62,968	124,391	802,629	1,857,954	2,466,510
ClientConnected: (ms)	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000
ClientBeginRequest: (ms)	0:00:23.766	0:00:46.312	0:00:00.000	0:00:23.422	0:00:51.266
GotRequestHeaders: (ms)	0:00:23.766	0:00:46.312	0:00:00.000	0:00:23.422	0:00:51.266
ClientDoneRequest: (ms)	0:00:23.766	0:00:46.312	0:00:00.000	0:00:23.422	0:00:51.266
FiddlerBeginRequest: (ms)	0:00:23.766	0:00:46.312	0:00:00.000	0:00:23.422	0:00:51.266
ServerGotRequest: (ms)	0:00:23.766	0:00:46.312	0:00:00.000	0:00:23.422	0:00:51.266
ServerBeginResponse: (ms)	0:00:23.812	0:00:46.391	0:00:00.438	0:00:24.391	0:00:52.516
GotResponseHeaders: (ms)	0:00:23.812	0:00:46.391	0:00:00.438	0:00:24.391	0:00:52.516
ServerDoneResponse: (ms)	0:00:23.812	0:00:46.391	0:00:00.438	0:00:24.406	0:00:52.547
ClientBeginResponse: (ms)	0:00:23.812	0:00:46.391	0:00:00.453	0:00:24.406	0:00:52.547
ClientDoneResponse: (ms)	0:00:23.812	0:00:46.391	0:00:00.453	0:00:24.406	0:00:52.547

Table 4.5: COMP - B Gzip compressed hybrid architecture pre-processed results

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body(KB)	0	0	0	0	0
Total Request Header (KB)	500	501	501	501	501
Received: Header (KB)	246	247	247	248	248
Received: Body (KB)	62,722	124,144	802,382	1,857,706	2,466,262
Total Response Header (KB)	62,968	124,391	802,629	1,857,954	2,466,510
ClientConnected: (ms)	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000
ClientBeginRequest: (ms)	0:00:25.906	0:00:50.250	0:01:06.485	0:00:00.000	0:00:24.843
GotRequestHeaders: (ms)	0:00:25.906	0:00:50.250	0:01:06.485	0:00:00.000	0:00:24.843
ClientDoneRequest: (ms)	0:00:25.906	0:00:50.250	0:01:06.485	0:00:00.000	0:00:24.843
FiddlerBeginRequest: (ms)	0:00:25.906	0:00:50.250	0:01:06.485	0:00:00.000	0:00:24.843
ServerGotRequest: (ms)	0:00:25.906	0:00:50.250	0:01:06.485	0:00:00.000	0:00:24.843
ServerBeginResponse: (ms)	0:00:25.938	0:00:50.344	0:01:06.891	0:00:00.984	0:00:26.109
GotResponseHeaders: (ms)	0:00:25.938	0:00:50.344	0:01:06.891	0:00:00.984	0:00:26.109
ServerDoneResponse: (ms)	0:00:25.938	0:00:50.344	0:01:06.891	0:00:01.000	0:00:26.125
ClientBeginResponse: (ms)	0:00:25.938	0:00:50.344	0:01:06.906	0:00:01.000	0:00:26.140
ClientDoneResponse: (ms)	0:00:25.938	0:00:50.344	0:01:06.906	0:00:01.000	0:00:26.140

Table 4.6: COMP - C Gzip hybrid architecture compressed pre-processed results

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body(KB)	0	0	0	0	0
Total Request Header (KB)	500	501	501	501	501
Received: Header (KB)	246	247	247	248	248
Received: Body (KB)	62,722	124,144	802,382	1,857,706	2,466,262
Total Response Header (KB)	62,968	124,391	802,629	1,857,954	2,466,510
ClientConnected: (ms)	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000
ClientBeginRequest: (ms)	0:00:05.703	0:00:27.235	0:00:00.000	0:01:18.578	0:01:58.485
GotRequestHeaders: (ms)	0:00:05.703	0:00:27.235	0:00:00.000	0:01:18.578	0:01:58.485
ClientDoneRequest: (ms)	0:00:05.703	0:00:27.235	0:00:00.000	0:01:18.578	0:01:58.485
FiddlerBeginRequest: (ms)	0:00:05.703	0:00:27.235	0:00:00.000	0:01:18.578	0:01:58.485
ServerGotRequest: (ms)	0:00:05.703	0:00:27.235	0:00:00.000	0:01:18.578	0:01:58.485
ServerBeginResponse: (ms)	0:00:05.750	0:00:27.297	0:00:00.437	0:01:19.547	0:01:59.766
GotResponseHeaders: (ms)	0:00:05.750	0:00:27.313	0:00:00.437	0:01:19.547	0:01:59.766
ServerDoneResponse: (ms)	0:00:05.750	0:00:27.313	0:00:00.453	0:01:19.563	0:01:59.781
ClientBeginResponse: (ms)	0:00:05.750	0:00:27.313	0:00:00.453	0:01:19.563	0:01:59.781
ClientDoneResponse: (ms)	0:00:05.750	0:00:27.313	0:00:00.453	0:01:19.563	0:01:59.781

Table 4.7: Average results of computers A, B, and C while Gzip compressed hybrid architecture

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500.00	501.00	501.00	501.00	501.00
Sent: Body(KB)	-	-	-	-	-
Total Request Header (KB)	500.00	501.00	501.00	501.00	501.00
Received: Header (KB)	246.00	247.00	247.00	248.00	248.00
Received: Body (KB)	62,722.00	124,144.00	802,382.00	1,857,706.00	2,466,262.00
Total Response Header (KB)	62,968.00	124,391.00	802,629.00	1,857,954.00	2,466,510.00
ClientConnected: (ms)	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000
ClientBeginRequest: (ms)	0:00:18.458	0:00:41.266	0:00:22.162	0:00:34.000	0:01:04.865
GotRequestHeaders: (ms)	0:00:18.458	0:00:41.266	0:00:22.162	0:00:34.000	0:01:04.865
ClientDoneRequest: (ms)	0:00:18.458	0:00:41.266	0:00:22.162	0:00:34.000	0:01:04.865
FiddlerBeginRequest: (ms)	0:00:18.458	0:00:41.266	0:00:22.162	0:00:34.000	0:01:04.865
ServerGotRequest: (ms)	0:00:18.458	0:00:41.266	0:00:22.162	0:00:34.000	0:01:04.865
ServerBeginResponse: (ms)	0:00:18.500	0:00:41.344	0:00:22.589	0:00:34.974	0:01:06.130
GotResponseHeaders: (ms)	0:00:18.500	0:00:41.349	0:00:22.589	0:00:34.974	0:01:06.130
ServerDoneResponse: (ms)	0:00:18.500	0:00:41.349	0:00:22.594	0:00:34.990	0:01:06.151
ClientBeginResponse: (ms)	0:00:18.500	0:00:41.349	0:00:22.604	0:00:34.990	0:01:06.156
ClientDoneResponse: (ms)	0:00:18.500	0:00:41.349	0:00:22.604	0:00:34.990	0:01:06.156

Table 4.8: Average results of computers A, B, and C while uncompressed hybrid architecture

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500.00	501.00	501.00	501.00	501.00
Sent: Body(KB)	-	-	-	-	-
Total Request Header (KB)	500.00	501.00	501.00	501.00	501.00
Received: Header (KB)	223.00	223.00	224.00	224.00	224.00
Received: Body (KB)	199,777.00	386,777.00	2,366,776.00	5,527,776.00	7,299,776.00
Total Response Header (KB)	200,000.00	387,000.00	2,367,000.00	5,528,000.00	7,300,000.00
ClientConnected: (ms)	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000
ClientBeginRequest: (ms)	0:00:26.979	0:00:10.151	0:00:31.750	0:00:54.755	0:01:14.911
GotRequestHeaders: (ms)	0:00:26.979	0:00:10.151	0:00:31.750	0:00:54.755	0:01:14.911
ClientDoneRequest: (ms)	0:00:26.979	0:00:10.151	0:00:31.750	0:00:54.755	0:01:14.911
FiddlerBeginRequest: (ms)	0:00:26.979	0:00:10.151	0:00:31.750	0:00:54.755	0:01:14.911
ServerGotRequest: (ms)	0:00:26.979	0:00:10.151	0:00:31.750	0:00:54.755	0:01:14.911
ServerBeginResponse: (ms)	0:00:27.015	0:00:10.203	0:00:32.026	0:00:55.391	0:01:15.740
GotResponseHeaders: (ms)	0:00:27.015	0:00:10.203	0:00:32.026	0:00:55.391	0:01:15.740
ServerDoneResponse: (ms)	0:00:27.021	0:00:10.214	0:00:32.052	0:00:55.442	0:01:15.807
ClientBeginResponse: (ms)	0:00:27.021	0:00:10.214	0:00:32.057	0:00:55.453	0:01:15.818
ClientDoneResponse: (ms)	0:00:27.021	0:00:10.214	0:00:32.062	0:00:55.458	0:01:15.833

4.4 EXPERIMENT PROCESSED DATA

4.4.1 COMPRESSED/UNCOMPRESSED RESPONSE MESSAGES OF HYBRID ARCHITECTURE

Processed data of uncompressed hybrid architecture SOAP responses messages are shown in Table 4.9 while those of compressed hybrid architecture SOAP response messages are shown in Table 4.10. This information was derived from Table 4.7 and Table 4.8 respectively.

Table 4.9: Hybrid architecture, uncompressed SOAP response messages

Original file size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500.00**	**501.00**	**501.00**	**501.00**	**501.00**
Sent: Body (KB)	**-**	**-**	**-**	**-**	**-**
Total Request Header (KB)	**500.00**	**501.00**	**501.00**	**501.00**	**501.00**
Received: Header (KB)	**223.00**	**223.00**	**224.00**	**224.00**	**224.00**
Received: Body (KB)	**199,777.00**	**386,777.00**	**2,366,776.00**	**5,527,776.00**	**7,299,776.00**
Total Response Header (KB)	**200,000.00**	**387,000.00**	**2,367,000.00**	**5,528,000.00**	**7,300,000.00**

Table 4.10: Hybrid architecture, compressed SOAP response message

Original file size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500.00**	**501.00**	**501.00**	**501.00**	**501.00**
Sent: Body (KB)	**-**	**-**	**-**	**-**	**-**
Total Request Header (KB)	**500.00**	**501.00**	**501.00**	**501.00**	**501.00**
Received: Header (KB)	**246.00**	**247.00**	**247.00**	**248.00**	**248.00**
Received: Body(KB)	**62,722.00**	**124,144.00**	**802,382.00**	**1,857,706.00**	**2,466,262.00**
Total Response Header	**62,968.00**	**124,391.00**	**802,629.00**	**1,857,954.00**	**2,466,510.00**

4.4.2 TIME TAKEN TO TRANSFER SOAP REQUEST/RESPONSE IN HYBID ARCHITECTURE

Using the average results shown in Table 4.7 and Table 4.8, we calculated the time taken to transfer SOAP requests using equation (4.1) whose information is shown in Table 4.11. The time taken to transfer a SOAP response was calculated using equation (4.2) (Fiddler, 2014). Improved time was calculated using equation (4.3) while the percentage improved time was calculated using equation (4.4) Table 4.12 shows the results.

$$\text{Time taken to transfer request} = (\text{ServerGotRequest} - \text{ClientBeginRequest}) \qquad (4.1)$$

Table 4.11: Time to transfer SOAP request of to the server for Hybrid Compressed and Uncompressed

Original file size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Hybrid Compressed (ms)	**0**	**0**	**0**	**0**	**0**
Hybrid Uncompressed (ms)	**0**	**0**	**0**	**0**	**0**

$$\text{Time taken to transfer SOAP responses} = (\text{ServerGotRequest} - \text{ClientDoneRequest}) - (\text{DNSTime} + \text{TCPConnectTime}) \qquad (4.2)$$

$$\text{Improved time} = \text{Hybrid \_uncompressed} - \text{Hybrid\_compressed} \qquad (4.3)$$

$$\text{Percentage improved time} = (\text{Improved\_time} \div \text{Hybrid \_uncompressed}) * 100 \qquad (4.4)$$

Table 4.12: Time taken to transfer compressed and uncompressed SOAP responses and percentage improvement time in the Hybrid architectures.

Original file size (KB)	**200**	**387**	**2367**	**5528**	**7300**	**Avg. % improvement**
Hybrid Compressed (ms)	**0**	**5**	**5**	**16**	**21**	
Hybrid Uncompressed (ms)	**5**	**11**	**27**	**52**	**68**	
Improved time (ms)	**5**	**6**	**22**	**36**	**47**	
Percentage improved time	**100**	**54.55**	**81.48**	**69.23**	**69.12**	**74.87**

4.4.3 TIME TAKEN TO PROCESS SOAP REQUEST

Time to process SOAP requests on the server was calculated using equation (4.5) (Fiddler, 2014). Degraded time was calculated using equation (4.6) while percentage degraded time was calculated using equation (4.7). Results are shown in shown in Table 4.13.

$$\text{Time to process request} = (\text{ServerBeginResponse} - \text{ServerGotRequest}) \tag{4.5}$$

$$\text{Degraded time} = \text{Hybrid\_compressed} - \text{Hybrid\_uncompressed} \tag{4.6}$$

$$\text{Percentage degraded time} = (\text{Degraded time} \div \text{uncompressed}) * 100 \tag{4.7}$$

Table 4.13: Time taken to process compressed and uncompressed SOAP request and percentage degraded time in the hybrid architectures

Original file size (KB)	200	387	2367	5528	7300	Avg. % degraded time
Hybrid Compressed (ms)	42	78	427	974	1266	
Hybrid Uncompressed (ms)	36	52	276	635	828	
Degraded time (ms)	6	26	151	339	438	
Percentage degraded time	16.67	50.00	54.71	53.39	52.90	45.53

4.5 SOAP PERFORMANCE EVALUATION METRICS

SOAP is a service oriented technology with Quality of Service (QoS) requirements. QoS can be defined as a set of techniques geared towards management of resources (CISCO, 2013). Moreover, QoS is a set of perceivable characteristics expressed in a user-friendly language with quantifiable parameters that may be subjective or objective. Software performance evaluation is an area of interest in QoS; which is also a field of concern in software engineering (Martinez, 2005).

Performance evaluation emerged in the 1970's and 1980's as an important component in computer science. It involves the use of accepted methods in measuring computer systems. The field of computer systems and software engineering focuses on specific components in computer science. These components can be evaluated in terms of their effectiveness. SOAP performance metrics involves a standard measure of SOAP performance indices. In the OSI (Open Standards Interconnection) model of communication in a network, SOAP metrics can be evaluated at multiple layers in the protocol stack, for instance, IP packets round trip time (network layer), and channel utilization (transport layer). There are other various performance evaluation parameters which include: throughput, good put, packet loss rate, and MAC layer retries (Martinez, 2005) (Tekli et al., 2011) (Bob, 2006) (Shariq et al., 2012).

This research is interested in SOAP performance which is widely measured in terms of:

(i) Round trip time (responsiveness): It is the time required to traverse a network and back. Round trip time is measured in milliseconds (ms).

(ii) Bandwidth/channel utilization: It measures utilization of a channel. It is the amount of data transmitted or received at a given time. It is measured in megabytes per second (mbps).

(iii) Throughput: Measures the packets that are flowing out (e.g. requests) of a node/client. In WS, it can be measured in either megabytes per second (mbps) or requests per second (req/sec). It is usually measured at the server-side.

We did not calculate bandwidth because, as described in experimental set up in section 4.1, bandwidth was set as a constant value of 10mbps. We further measured SOAP compression using compression ratio percentage (Mark et al., 1995). We therefore measure SOAP performance using the following metrics:

(i) Compression ratio percentage

(ii) Round trip time (responsiveness)

(iii)Throughput

4.5.1 COMPRESSION RATIO PERCENTAGE

Compression ratio percentage is used as tool for analysis to measure percentage effect of compression. Compression percentage means that for a file that doesn't change at all when compressed, it will have a compression ratio of zero percent. A file compressed down to a quarter of its original size will have a compression ratio of 75 percent. A file that shrinks down to 0 bytes will have a compression ratio of 100 percent. This way of measuring compression may not be perfect, but it shows perfect compression at 100 percent and total compression failure at zero (0) percent. In fact, a file that goes through a compression program and comes out larger than its original size will have negative compression ratio percentage. The formula of calculating compression ratio percentage is show in equation (4.8) (Mark et al., 1995).

$$(1 - (\text{compressed\_size} / \text{raw\_size})) * 100 \tag{4.8}$$

Table 4.9 and Table 4.10 highlighted SOAP messages file sizes. From this, the total size of SOAP response was calculated by adding the size of the header and the size of the body of the SOAP response message using equation (4.9). The results of compression ratio percentage of SOAP messages responses are evaluated as shown in Table 4.14.

$$\text{Total size of SOAP response} = \text{size of response header} + \text{size of response body} \tag{4.9}$$

Table 4.14: Compression ratio percentage SOAP messages of Hybrid Compressed and Uncompressed

Methodology	Original file size (KB)	200	387	2367	5528	7300
Response Hybrid Uncompressed	Received: Header (KB)	223.00	223.00	224.00	224.00	224.00
	Received: Body (KB)	199,777.00	386,777.00	2,366,776.00	5,527,776.00	7,299,776.00
	Total Response Header (KB)	200,000.00	387,000.00	2,367,000.00	5,528,000.00	7,300,000.00
Response Hybrid Uncompressed	Received: Header (KB)	246.00	247.00	247.00	248.00	248.00
	Received: Body(KB)	62,722.00	124,144.00	802,382.00	1,857,706.00	2,466,262.00
	Total Response Header (KB)	62,968.00	124,391.00	802,629.00	1,857,954.00	2,466,510.00

We calculated SOAP response messages compression ratio percentage and average compression ratio percentage using equation (4.10) and equation (4.11) respectively as shown in Table 4.15.

(1 - (ASP_compressed_response_size ÷ ASP_uncompressed_response_size)) * 100 (4.10)

Average compression ratio% = Sum of compression ratio % ÷ No. of compressed samples (4.11)

Table 4.15: Compression ratio percentage for hybrid compressed and uncompressed SOAP messages

Original file size (KB)	200	387	2367	5528	7300	Avg. compression ratio %
Response hybrid uncompressed size (KB)	200,000	387,000	2,367,000	5,528,000	7,300,000	
Response hybrid compressed size (KB)	62,968	124,391	802,629	1,857,954	2,466,510	
Compression ratio %	68.516	67.8576	66.0909	66.3901	66.2122	67.0134

4.5.2 ROUND TRIP TIME (RTT)

Round trip time (RTT) is the time required to traverse a network and back. Round trip time is measured in milliseconds (Tekli et al., 2011). This research's RTT is the time taken for a SOAP message to traverse a network and come back to the client.

Table 4.11 gave all SOAP requests transfer time as zero. We therefore evaluated RTT using time taken to transfer SOAP response and time taken to process SOAP requests as show in equation

RTT = Time taken to transfer SOAP response + Time taken to process SOAP request (4.12)

(4.12). Degraded RTT was calculated using equation (4.13). Equation (4.14) was used to calculate percentage degraded RTT. Results as are show in Table 4.16.

Degraded RTT = Hybrid_Compressed – Hybrid_Uncompressed (4.13)

Percentage RTT degraded = (Degraded RTT ÷ Hybrid_Uncompressed) * 100 (4.14)

Table 4.16: Round trip time results for Hybrid Compressed and Uncompressed

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**	**Avg. % RTT degraded**
Hybrid Compressed (ms)	**42**	**83**	**432**	**990**	**1287**	
Hybrid Uncompressed (ms)	**41**	**63**	**303**	**687**	**896**	
Degraded RTT (ms)	**1**	**20**	**129**	**303**	**391**	
Percentage RTT degraded	**2.44**	**31.75**	**42.57**	**44.10**	**43.69**	**32.91**

4.5.3 THROUGHPUT

Throughput measures the packets that flow out (e.g. requests) of a node/client. In WS, throughput can be measured in either megabytes per second (mbps) or requests per second (req/sec). Throughput is usually measured at the server-side (Tekli et al., 2011). In this research throughput is measured in requests per second using equation (4.15). The average throughput was measure by dividing total number of request by total RTT of these requests as shown in equation (4.16); Table 4.17 and Table 4.18 captured the results.

Throughput = Request ÷ RTT (4.15)

Average throughput = Average total number of request ÷ Average total RTT (4.16)

Table 4.17: Average throughput performance measures for hybrid compressed

File Size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Average number of requests	**1**	**1**	**1**	**1**	**1**
Average Hybrid Compressed RTT (s)	**0.042**	**0.083**	**0.432**	**0.990**	**1.287**
Average throughput (request per second)	**23**	**12.05**	**2.31**	**1.01**	**0.78**

Table 4.18: Average throughput performance measures for hybrid uncompressed

Average file size in KB	**200**	**387**	**2367**	**5528**	**7300**
Average number of requests	**1**	**1**	**1**	**1**	**1**
Average Hybrid Uncompressed RTT (s)	**0.041**	**0.063**	**0.303**	**0.687**	**0.896**
Average throughput (request per second)	**24.39**	**15.87**	**3.3**	**1.46**	**1.12**

4.6 RESULTS

Results in this section are guided by the processed and evaluated data outlined in section 4.3 and section 4.5 respectively. We noted down results as follows: compression ratio, time to transfer SOAP messages, time to process requests, round trip time, and throughput.

4.6.1 RESULTS FOR COMPRESSION RATIO PERCENTAGES

Figure 4.3 contains a graph illustrating change in compression ratio percentage against file size of SOAP response. This graph was derived from Table 4.15 section 4.5.1. It depicts the percentage change in compression ratio percentage against change in file sizes between 200KB and 7300KB. File size 2367KB recorded the lowest compression ratio percentage.

The general trend of this graph shows the line sloping to the right. This indicated that smaller file sizes exhibited better compression ratios than large file sizes after being compressed with Gzip algorithm.

There was substantial relationship between file size and compression ratio percentage. This is shown by sloping of the line representing the change in compression ratio percentage. We note that file size had a notable negative relationship with compression ratio percentage.

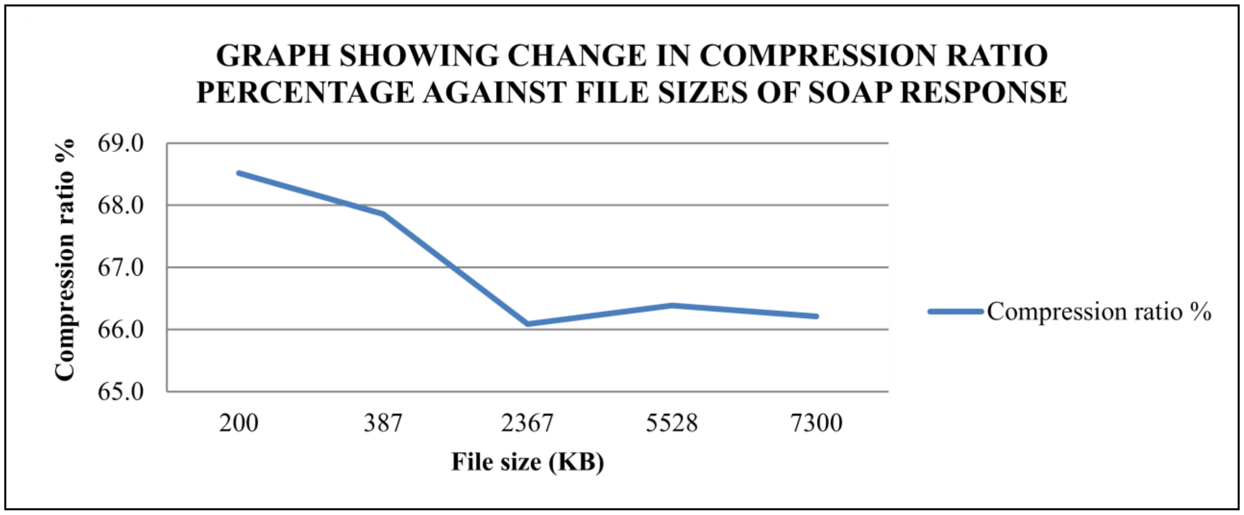


Figure 4.3: Change in compression ratio percentage against file sizes of SOAP response

4.6.2 RESULTS FOR TIME TO TRANSFER SOAP MESSAGES

(i) Transfer time of compressed and uncompressed hybrid architecture SOAP requests

Figure 4.4 represents a graph depicting change in transfer time against file size of hybrid compressed and hybrid uncompressed SOAP requests. This graph was as a result of data from Table 4.12. We considered time to transfer response from the server to the client. Time to transfer requests from the client to the server was left out because it gave zero values as show in Table 4.11. The graph indicates the change in transfer time against file sizes between 200KB and 7300KB for hybrid uncompressed and hybrid compressed SOAP messages.

It is observed that, hybrid compressed recorded the same time value for 387KB and 2367KB file sizes. Nevertheless, the trend of the graph shows that both hybrid architectures, compressed and uncompressed lines rise steadily. Indicating that smaller file sizes portray smaller transfer time than larger files. The line representing hybrid compressed runs below hybrid uncompressed. This indicated that uncompressed take much longer time to transfer files compared to compressed files.

There was a notable relationship between file size and time taken to transfer SOAP request. This was indicated by steady rising of the lines representing both hybrids, compressed and uncompressed. So far we say that file size had a notable positive relationship with time.

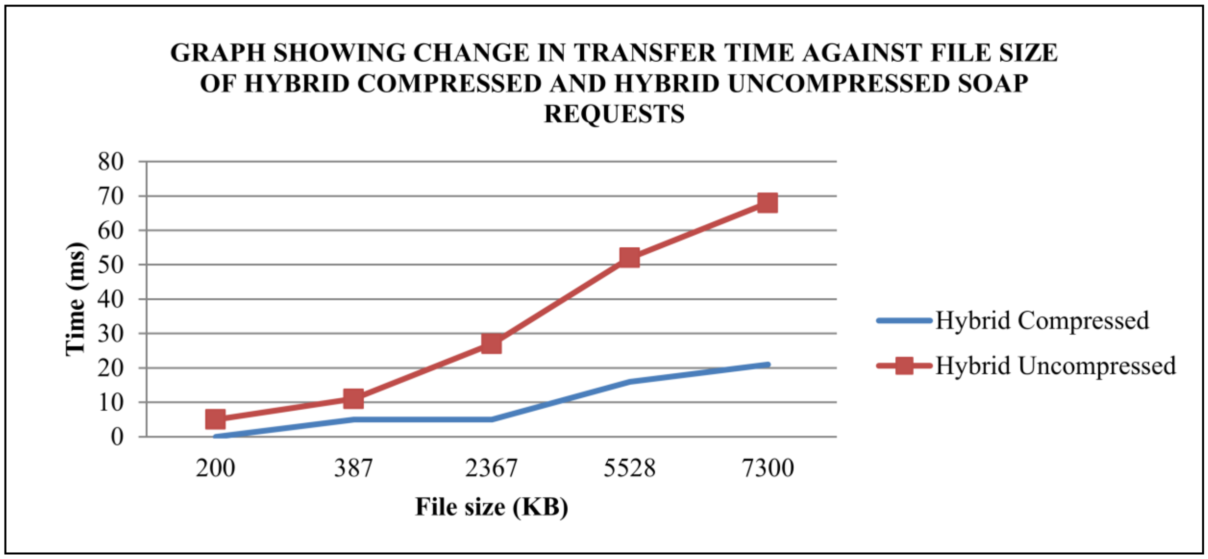


Figure 4.4: Change in transfer time against file size of compressed and uncompressed SOAP requests of the Hybrid Architecture

(ii) Percentage change in transfer time of compressed and uncompressed hybrid architectures SOAP requests

Figure 4.5 represents a graph showing change in percentage improved transfer time against file size of hybrid compressed compared to hybrid uncompressed SOAP requests. This graph is as a result of data from Table 4.12. The graph indicates the change in percentage improved transfer time against file sizes between 200KB and 7300KB for hybrid uncompressed and hybrid compressed SOAP messages.

The graph depicts that, file size 387KB recorded the lowest percentage improved transfer time while 5528KB and 7300KB recorded close percentages values the former being slightly higher. However, the general trend of the graph shows that the line slopes steadily to the right. This indicated that smaller file sizes portray a higher percentage improvement in transfer time than larger files.

There was notable relationship between file size and change in percentage improved transfer time. This is indicated by steady slope of the lines to the right representing percentage change transfer time. We note that file size had a notable negative relationship with percentage improvement transfer time.

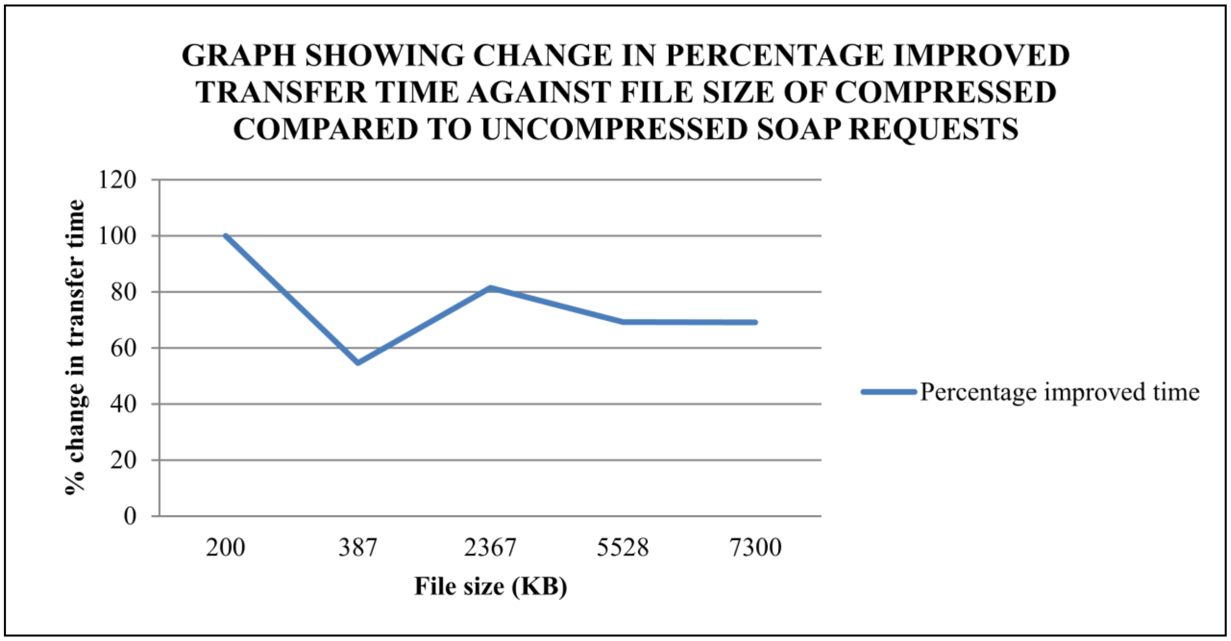


Figure 4.5: Change in percentage improved transfer time against file size of compressed compared to uncompressed SOAP requests of the Hybrid Architecture

(iii) Transfer time of compressed hybrid architecture and Seyyed et al. (2011) SOAP requests

Let's compare compression and Seyyed et al. (2011) in terms of time to transfer SOAP messages. To measure improvement in transfer time and percentage improvement, we employed equation (4.17) and equation (4.18) respectively. The results were tabulated in Table 4.19.

$$\text{Improved transfer time} = \text{Seyyed\_et\_al\_2011} - \text{Hybrid\_Compressed} \qquad (4.17)$$

$$\text{Percentage improvement} = (\text{Improved\_transfer\_time} \div \text{Seyyed\_et\_al\_2011}) * 100 \qquad (4.18)$$

Table 4.19: Transfer time and percentage improvement of compared to Seyyed et al. (2011) of SOAP messages of the Hybrid Architecture

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**	**Avg. % improvement**
Hybrid Compressed	**0**	**5**	**5**	**16**	**21**	
Seyyed et al. (2011)	**2**	**6**	**83**	**83**	**107**	
Improved time	**2**	**1**	**78**	**67**	**86**	
Percentage improvement	**100**	**166.67**	**93.97**	**80.72**	**80.37**	**104.35**

Figure 4.6 represents a line graph depicting transfer time against file size of hybrid compressed and Seyyed et al. (2011) SOAP messages. This graph was as a result of data from Table 4.19. It is observed that, compressed hybrid architecture recorded the same time value for 387KB and 2367KB file sizes. Seyyed et al. (2011) also recorded same transfer value of 2367KB and 5528KB file sizes. Nevertheless, the trend of the graph shows that both compressed hybrid architecture and Seyyed et al. (2011) lines rose steadily. Indicating that smaller file sizes recorded smaller transfer time than larger files. The line representing compressed hybrid

architecture runs below Seyyed et al. (2011). This indicated that hybrid compressed took less time to transfer files.

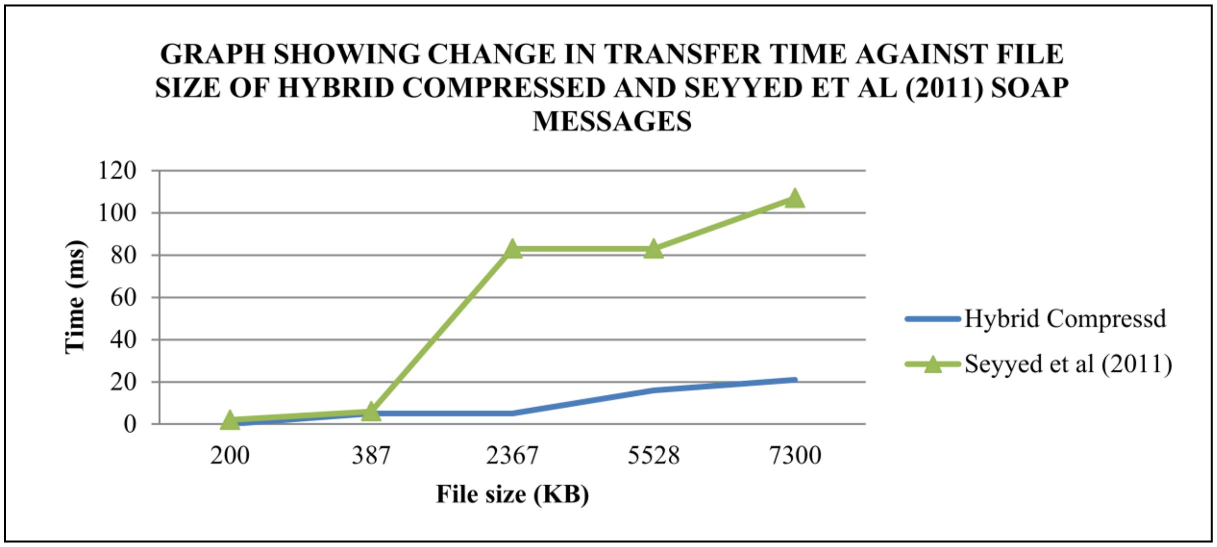


Figure 4.6: Change in transfer time against file size of hybrid compressed and Seyyed et al. (2011) SOAP messages

Figure 4.7 contains a graph showing change in percentage improvement against file size of compressed hybrid architecture compared to Seyyed et al. (2011). This graph was as a result of data from Table 4.19. It is clear that, 5528KB and 7300KB recorded slightly different percentage improvement values 7300 being slightly lower. File size 387KB recorded the highest percentage improvement value. The trend of the graph shows that the line sloped steadily to the right. This indicated that smaller files record better percentage improvement than larger files.

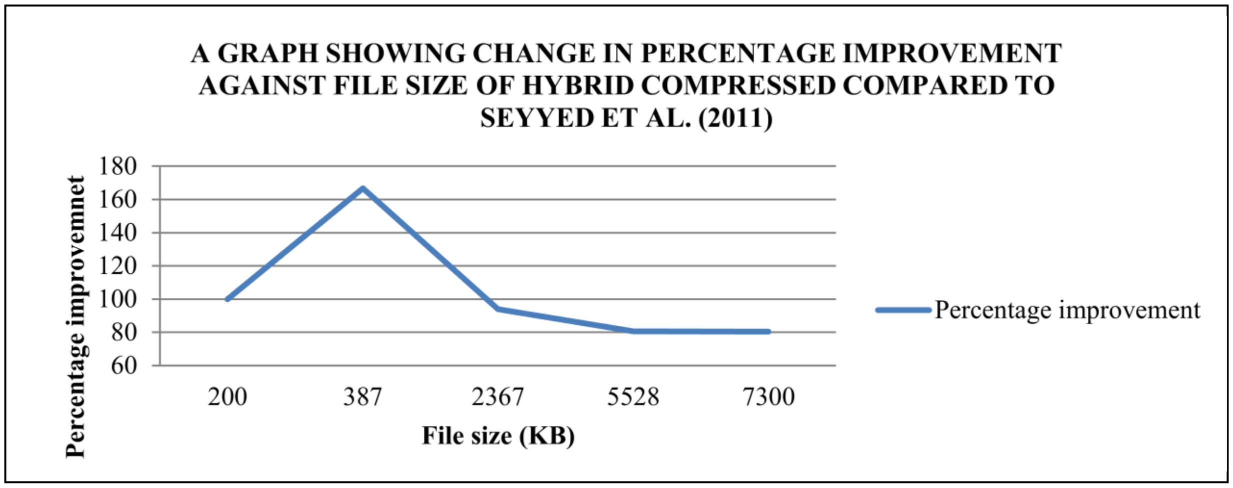


Figure 4.7: Change in percentage improvement against file size of hybrid compressed compared to Seyyed et al. (2011)

4.6.3 RESULTS FOR TIME TO PROCESS REQUEST

(i) Time to process requests of compressed and uncompressed hybrid architectures

Figure 4.8 illustrates a graph representing change in time to process SOAP request against change in file size of compressed and uncompressed. This graph was retrieved from Table 4.16 which shows results of time to process compressed and uncompressed request between 200KB and 7300KB file sizes.

The graph shows the line representing change in hybrid compressed and hybrid uncompressed rising moderately. This points out that that an increase in the file size results to an increase in the time to process a SOAP request for both hybrid architectures, compressed and uncompressed. Nevertheless, uncompressed line runs below compressed. This depicts that compressed hybrid architecture take more time to process as compared to uncompressed hybrid architecture. This is significant in the sense that compressed files in hybrid architecture take more time to be processed in the server-side.

There is a notable relationship between file size and time. This is indicated by the lines representing both hybrid architectures, compressed and uncompressed rising moderately. We can therefore say that file size had a notable positive relationship with time.

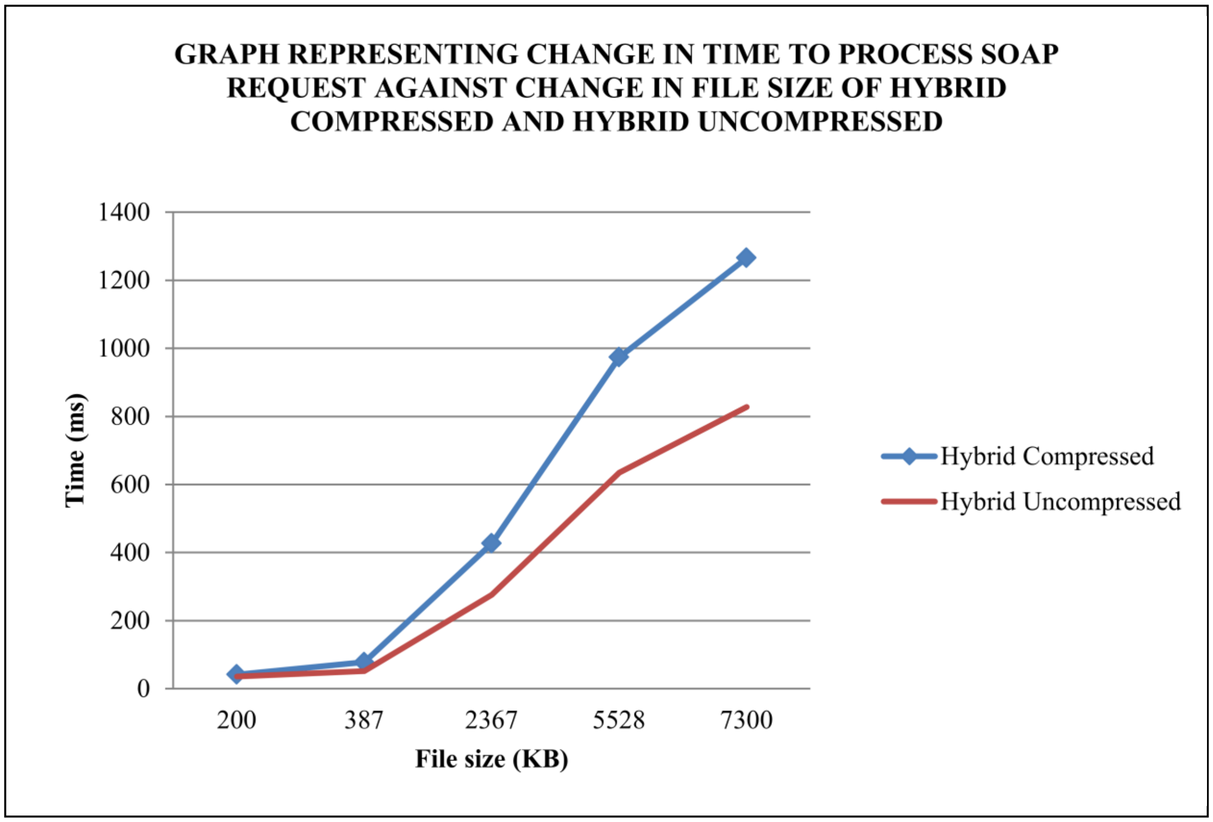


Figure 4.8: Change in time to process SOAP request against change in file size of compressed and uncompressed

(ii) Percentage change in processing time of SOAP request of compressed and uncompressed hybrid architectures

Figure 4.9 represents a graph representing change in percentage degraded time to process SOAP request against change in file size of compressed hybrid architecture compared to uncompressed hybrid architecture. This graph is as a result of data from Table 4.13.The graph indicates the change in percentage degraded time to process SOAP against file sizes between 200KB and

7300KB. The graph illustrates that the percentage degraded time rises steadily to file size 2367KB. Thereafter the percentage degraded time starts dropping slowly.

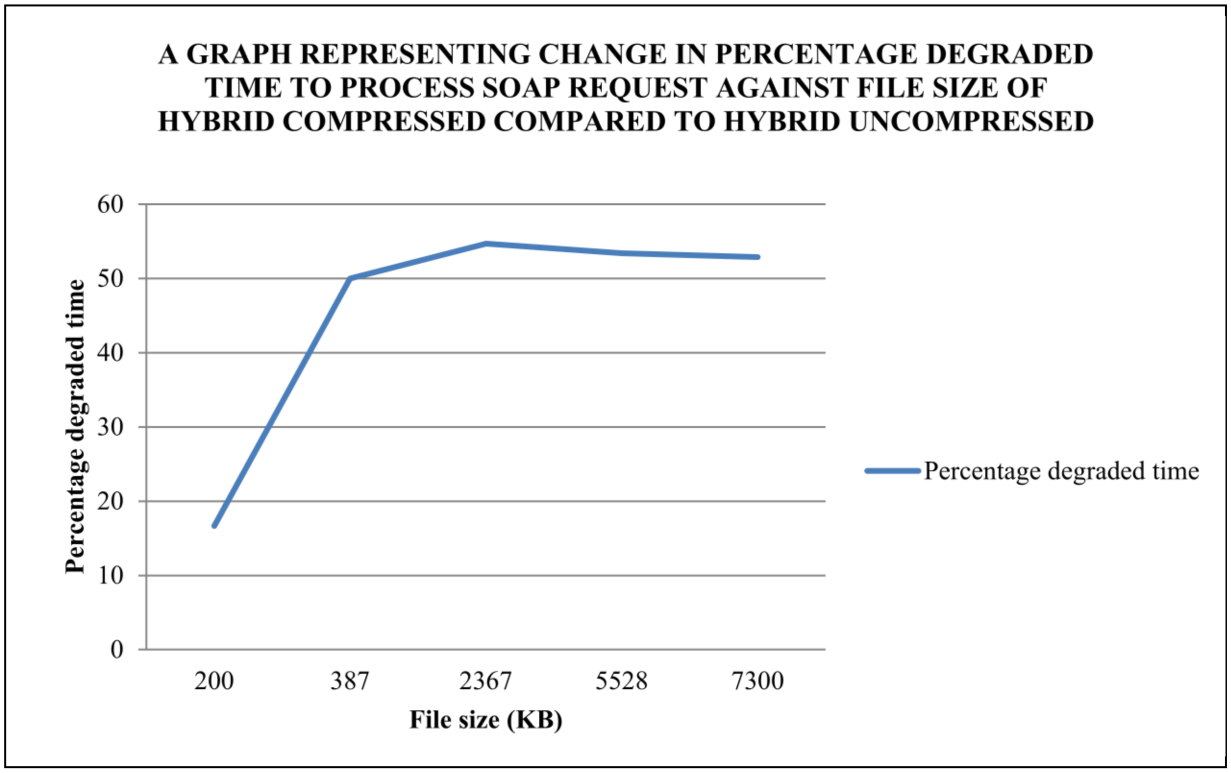


Figure 4.9: Change in percentage degraded time to process SOAP request against change in file size of compressed compared to uncompressed hybrid architectures

(iii) Time to process compressed hybrid architecture and Seyyed et al. (2011) SOAP requests

We compared compressed hybrid architecture and Seyyed et al. (2011) in terms of time to process SOAP request. To measure improved processing time and percentage improvement, we used equation (4.19) and equation (4.20) respectively. The results were tabulated in Table 4.20.

$$\text{Improved processing time} = \text{Seyyed\_et\_al\_2011} - \text{Hybrid\_Compressed} \qquad (4.19)$$

$$\text{Percentage improvement} = (\text{Improved processing time} \div \text{Seyyed\_et\_al\_2011}) * 100 \qquad (4.20)$$

Table 4.20: Processing time and percentage improvement of Hybrid Compressed compared to Seyyed et al. (2011) of SOAP messages

File size in KB	200	387	2367	5528	7300	Avg. % improvement
Hybrid Compressed	42	78	427	974	1266	
Seyyed et al. (2011)	129	227	813	1217	1400	
Improved time	87	149	386	243	134	
Percentage improvement	67.44	65.63	47.44	19.97	9.57	42.01

Figure 4.16 shows a graph representing change in time to process SOAP request against file size of compressed hybrid architecture and Seyyed et al. (2011). This graph is as a result of data from Table 4.20. The trend of the graph shows that compressed hybrid architecture and Seyyed et al. (2011) lines rise steadily. This means that smaller file sizes records smaller processing time than larger files. The line representing compressed hybrid architecture runs below Seyyed et al. (2011). This indicates that compressed took less time to process SOAP requests.

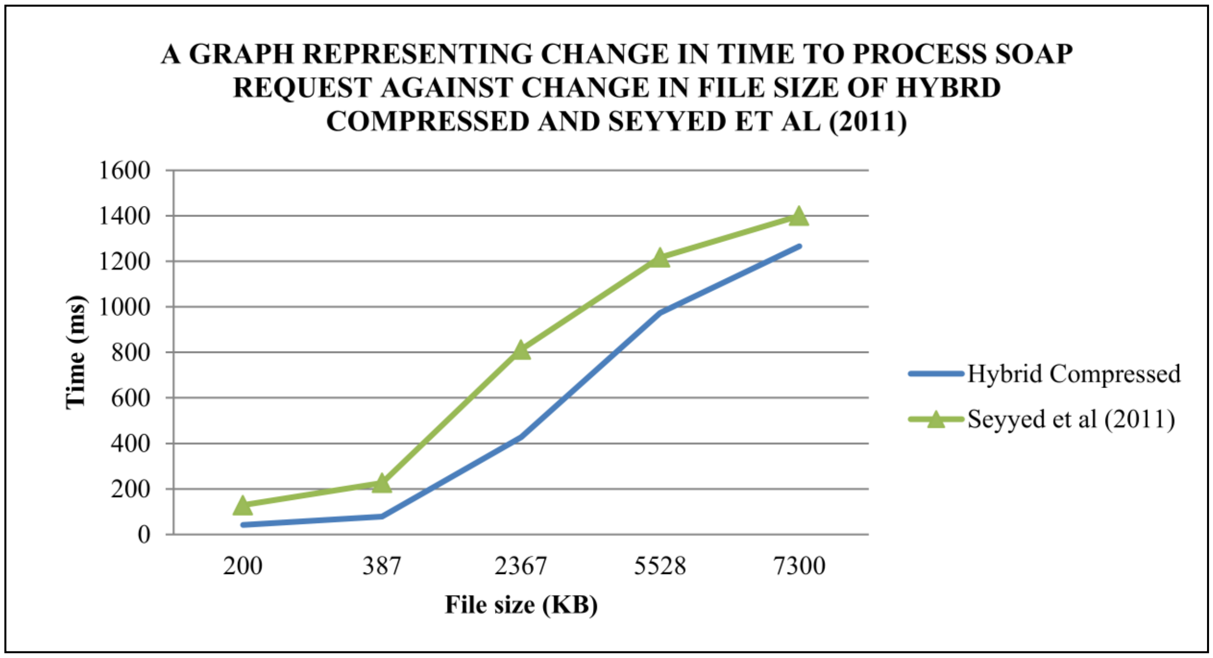


Figure 4.10: Change in time to process SOAP request against file size of hybrid compressed and Seyyed et al. (2011)

Figure 4.11 contains a line graph showing change in percentage improvement against file size of compressed hybrid architecture compared to Seyyed et al. (2011). This graph is as a result of data from Table 4.19. The trend of the graph shows that the line sloped steadily to the right. This indicates that smaller files record better percentage improvement in processing SOAP request than larger files.

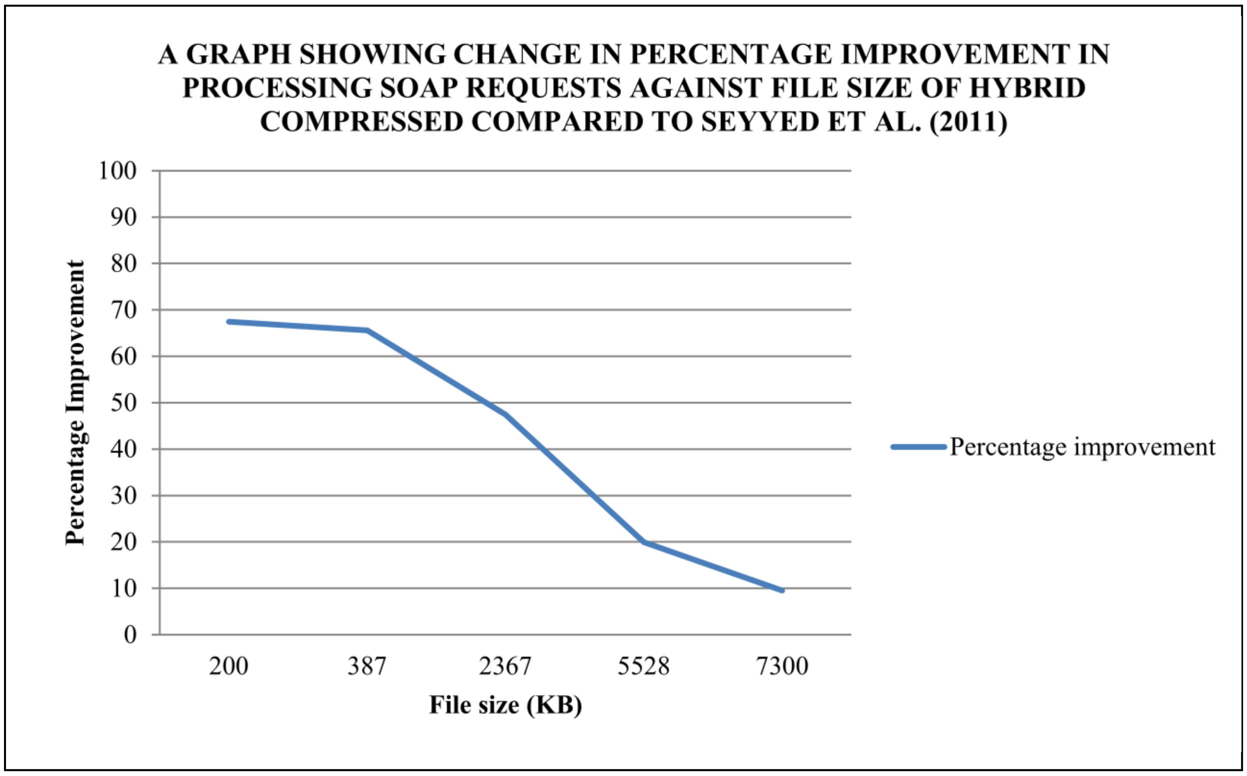


Figure 4.11: Change in percentage improvement in processing SOAP requests against file size of hybrid compressed compared to Seyyed et al. (2011)

4.6.4 RESULTS FOR ROUND TRIP TIME (RTT)

(i) Round trip time of compressed and uncompressed SOAP requests in the hybrid architecture

Figure 4.12 demonstrate a line graph describing change in round trip time against file sizes of compressed and uncompressed hybrid architectures. This graph is derived from Table 4.16. The file sizes range from 200KB to 7300KB.

The general trend of the graph shows that all the lines rise steadily. This indicates that large files exhibit more RTT compared to smaller files. Moreover, the line representing compressed hybrid architecture runs higher than uncompressed hybrid architecture; this indicates that compressed files recorded higher RTT than uncompressed.

There is notable relationship between file size and RTT. This is indicated by steady rise of the lines representing both compressed and uncompressed hybrid architectures. We generally point out that file size had a notable positive relationship with RTT as compressed hybrid architecture recorded higher RTT than uncompressed hybrid architecture.

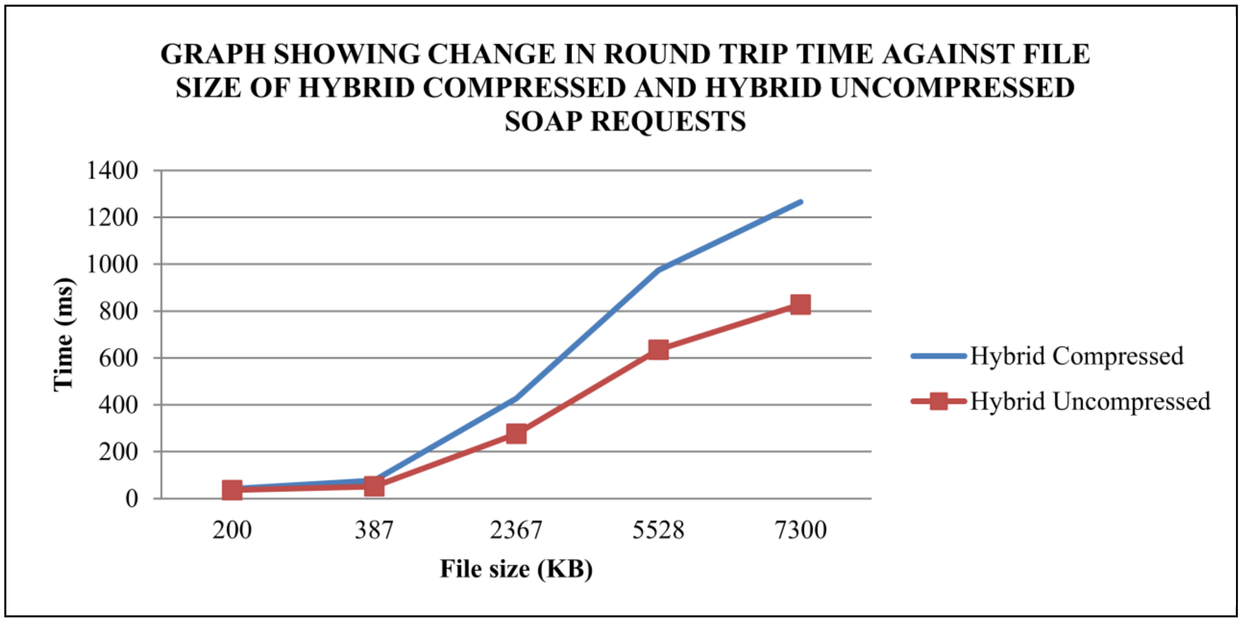


Figure 4.12: Change in round trip time against file sizes of compressed and uncompressed requests

(ii) Percentage change in round trip time of compressed and uncompressed SOAP requests in hybrid architecture

Figure 4.13 illustrates a line graph showing change in percentage degraded round trip time against file size of compressed compared to uncompressed SOAP requests in hybrid architectures. This graph is as result of data from Table 4.16. The graph indicates the change in degraded percentage round trip time against file sizes between 200KB and 7300KB for uncompressed and compressed SOAP messages in the hybrid architecture.

The graph shows that, the percentage change in degraded RTT rises steadily 200KB to 5528KB. It then falls slightly between 5528KBKB and 7300KB. This indicates that smaller compressed files recorded a smaller percentage degraded RTT compared to larger files.

There is notable relationship between file size and RTT. This is indicated by the steady rising of the lines representing percentage change in degraded RTT of compressed compared to uncompressed hybrid architectures. We can generally point out that file size had a notable positive relationship with percentage degraded RTT of compressed hybrid architecture for file sizes between 200KB and 7300KB.

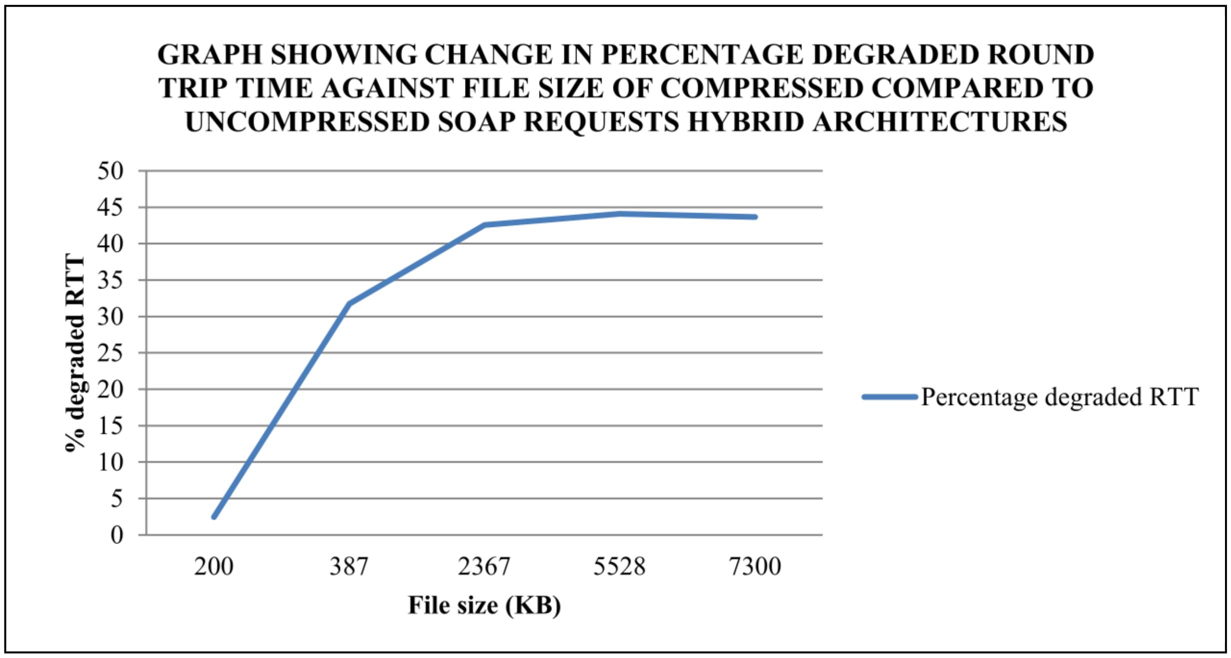


Figure 4.13: Change in percentage degraded round trip time against file size of compressed compared to uncompressed SOAP requests the hybrid architectures

(iii) Round trip time of compressed hybrid architecture and Seyyed et al. (2011) of SOAP requests

Let's compare compressed hybrid architecture and Seyyed et al. (2011) in terms of round trip time of SOAP messages. To measure improved round trip time and percentage improvement, we used equation (4.21) and equation (4.22) respectively. The results are presented in Table 4.21.

$$\text{Improved RTT} = \text{Seyyed\_et\_al\_2011} - \text{Hybrid\_Compressed} \tag{4.21}$$

$$\text{Percentage improved RTT} = (\text{Improved\_RTT} \div \text{Seyyed\_et\_al\_2011}) * 100 \tag{4.22}$$

Table 4.21: Round trip time and percentage improvement of compressed hybrid architecture compared to Seyyed et al. (2011) of SOAP messages

File size in KB	**200**	**387**	**2367**	**5528**	**7300**	**% RTT improved**
Hybrid Compressed (ms)	**42**	**83**	**432**	**990**	**1287**	
Seyyed et al. (2011) (ms)	**131**	**233**	**854**	**1300**	**1507**	
Improved RTT (ms)	**89**	**150**	**422**	**310**	**220**	
Percentage RTT improved	**67.94**	**64.38**	**49.41**	**23.85**	**14.60**	**44.04**

Figure 4.14 represents a line graph depicting transfer time against file size of compressed hybrid architecture and Seyyed et al. (2011) of SOAP messages. This graph is as a consequent of data from Table 4.21. The general trend of the graph shows that compressed hybrid architecture and Seyyed et al. (2011) lines rise steadily indicating that smaller file sizes records smaller transfer time than larger files. Nevertheless, the line representing compressed hybrid architecture runs below Seyyed et al. (2011). This indicates that compressed hybrid architecture took less time to transfer files.

There is notable relationship between file size and RTT. It is depicted by the steady rising of the lines representing change in RTT of compressed hybrid architecture and Seyyed et al. (2011). We generally note that file size had a notable positive relationship with RTT of compressed hybrid architecture and Seyyed et al. 2011 for file sizes between 200KB and 7300KB.

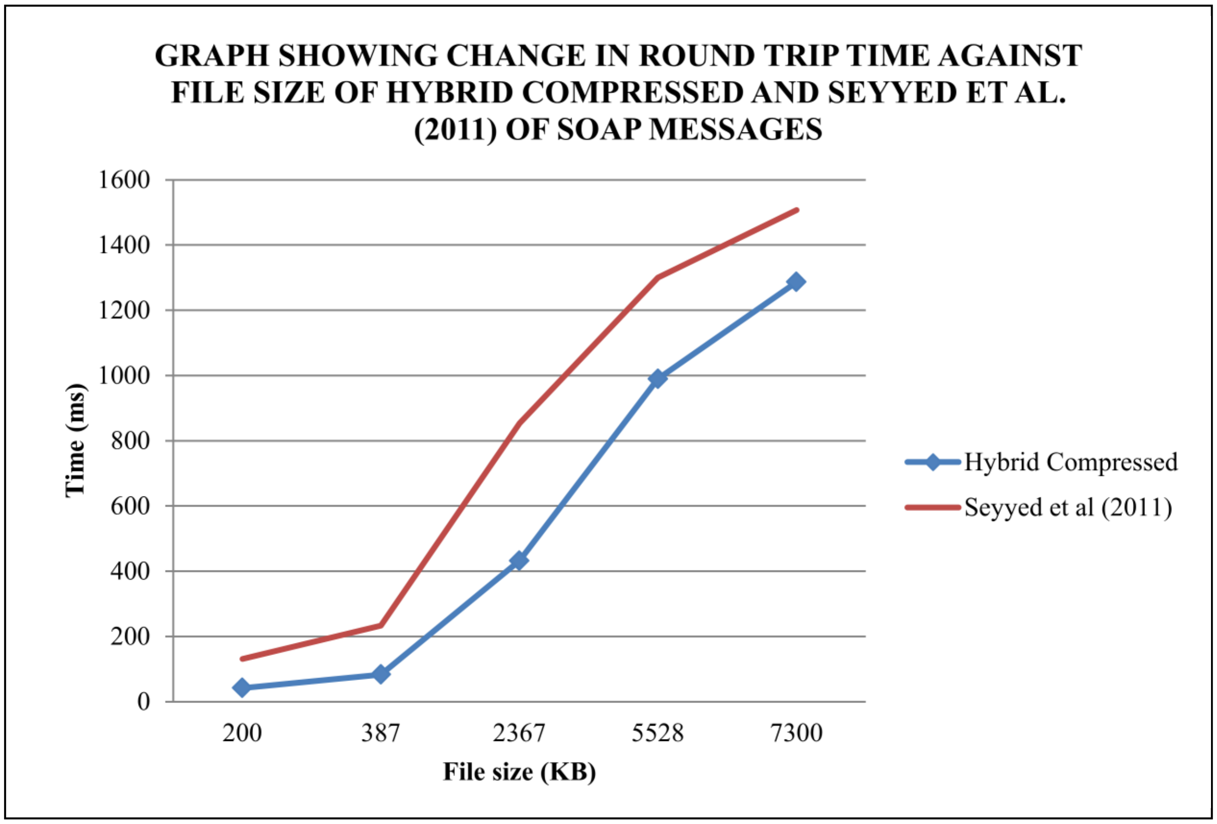


Figure 4.14: Change in round trip time against file size of compressed and Seyyed et al. (2011) of SOAP messages

Figure 4.15 illustrates a line graph showing change in percentage improved RTT against file size of compressed hybrid architecture compared to Seyyed et al. (2011) in SOAP messages. This graph is as a result of data from Table 4.21. The trend of the graph shows that the line slopes steadily to the right. This indicates that smaller file record better percentage improvement than larger files.

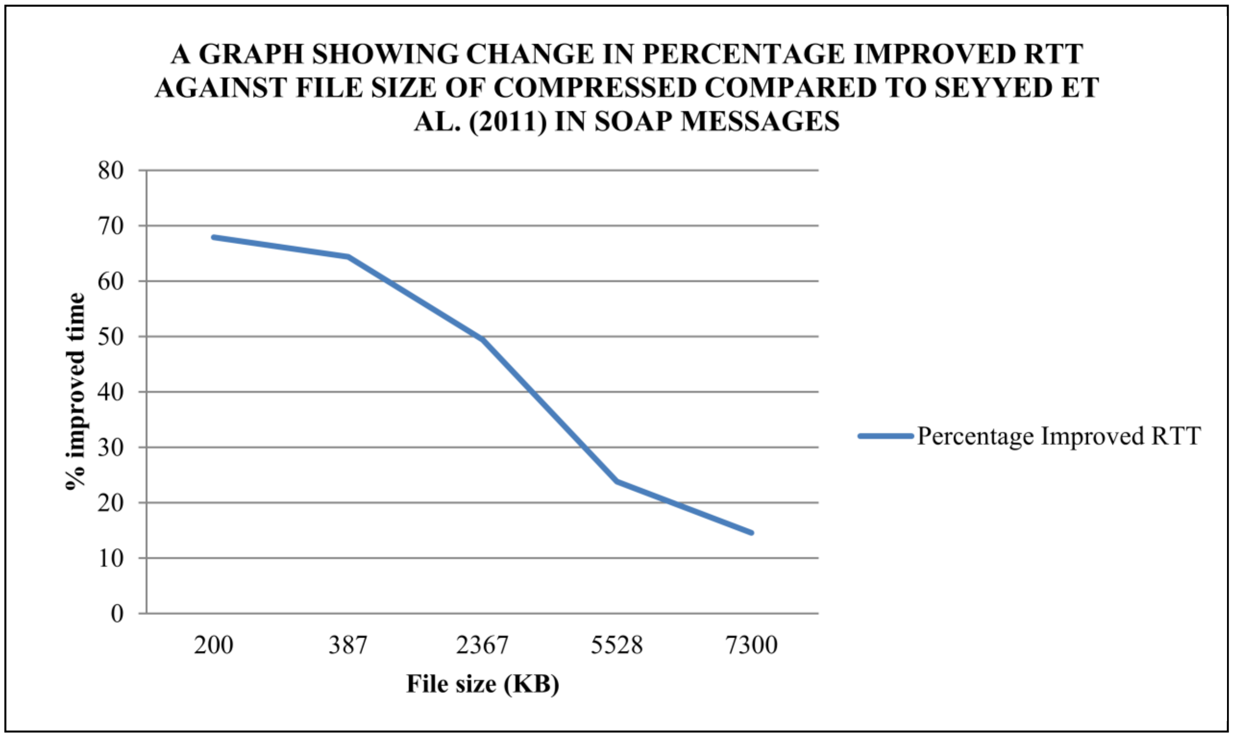


Figure 4.15: Change in percentage improved RTT against file size of hybrid compressed compared to Seyyed et al. (2011) in SOAP messages

4.6.5 RESULTS FOR THROUGHPUT

Table 4.22 contains an aggregation of throughput results of compressed hybrid architecture, uncompressed hybrid architecture and Seyyed et al. (2011). Figure 4.16 illustrates these data.

Table 4.22: An aggregation of throughput results of hybrid compressed, hybrid uncompressed and Seyyed et al. (2011)

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Hybrid Compressed (req/sec)	**23**	**12.05**	**2.31**	**1.01**	**0.78**
Hybrid Uncompressed (req/sec)	**24.39**	**15.87**	**3.3**	**1.46**	**1.12**
Seyyed et al. (2011) (req/sec)	**7.63**	**4.29**	**1.17**	**0.77**	**0.66**

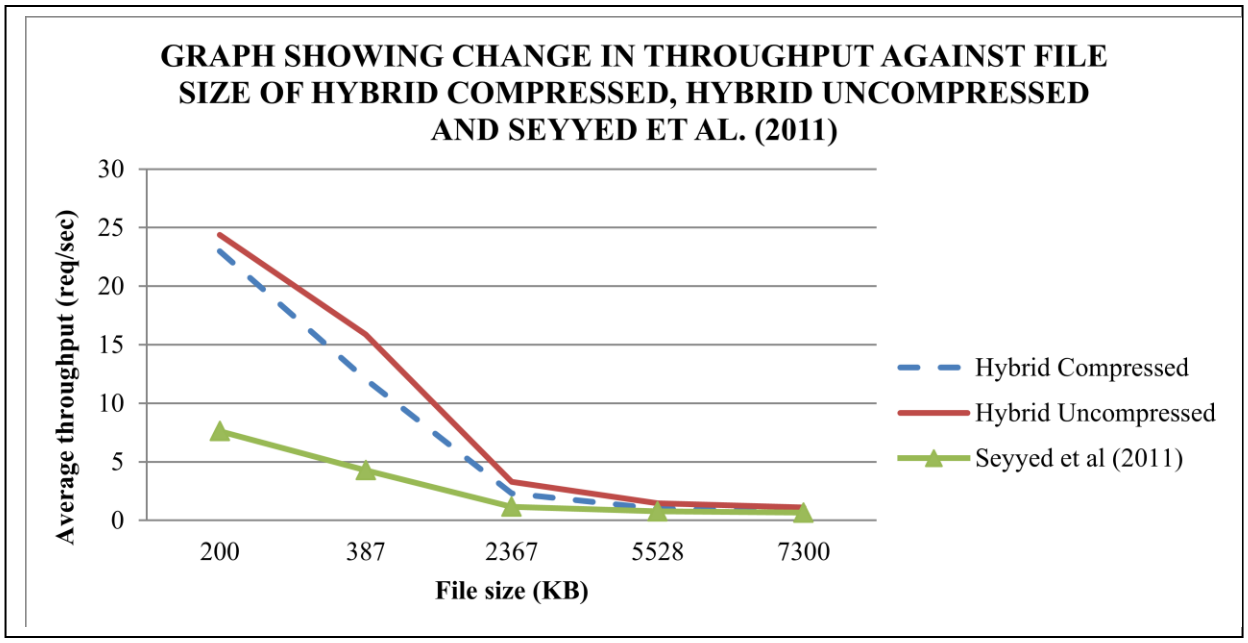


Figure 4.16: Change in throughput against change in file size of compressed, uncompressed and Seyyed et al. (2011)

Figure 4.16 illustrates a line graph outlining change in throughput against in file size of compressed hybrid architecture, uncompressed hybrid architecture and Seyyed et al. (2011). This graph was derived from Table 4.22, average throughput values. The file sizes range from 200KB to 7300KB.

The general trend of the graph shows that lines representing compressed hybrid architecture, uncompressed hybrid architecture, and Seyyed et al. (2011) slope to the right. This indicates that smaller file sizes exhibit better throughput values compared to large file sizes. Nevertheless, from the graph, hybrid compressed rides slightly below hybrid uncompressed and Seyyed et al. (2011) rides below hybrid compressed. This shows that hybrid uncompressed exhibits better throughput values than hybrid compressed which portray better throughput values than Seyyed et al. (2011).

There is notable relationship between file size and throughput. This is ascertained by the steady sloping to the right of the lines representing hybrid compressed, hybrid uncompressed and Seyyed et al. (2011). We can therefore say that file size has a notable negative relationship with throughput.

4.7 DISCUSSION

4.7.1 DISCUSIONS OF RESULTS FOR COMPRESSION RATIO PERCENTAGE

When interpreting and discussing compression ratio percentage, we should note that 100 percent compression ratio means perfect compression while zero percent compression means total compression failure (Mark et al., 1995). Poor compression ratios will have poor percentage compression ratios. Section 4.6.1 discussed the results illustrated in Figure 4.3. We observed that the line representing percentage compression ratio sloped steadily to the right. This means that smaller files exhibited better percentage compression ratio than larger files. From the graph we also infer that, the larger the file size the poor the percentage compression ratio. However, file size 2367KB recorded the lowest percentage compression ratio. The average compression ratio percentage was 67.01% as show in Table 4.15.

The lowest percentage compression ratio recorded by file size 2367KB could be due to the nature of data subjected to compression by Gzip algorithm. Tomasz et al. (2010) and Alex et al. (2008) noted that lossless compression algorithms work best in highly redundant data. Lossless compression algorithms exploit statistical redundancy to represent sender's data. Gzip is an example of lossless compression algorithm. SOAP messages are represented in XML. XML is highly redundant as it has redundant textual characteristics and it uses tags to delimit data (Snell et al., 2001) (Kohlhoff et al., 2003) (Seyyed et al., 2011). Nevertheless, Tomasz et al. (2010) and Alex et al. (2008) noted that highly entropic i.e. highly disordered data results to poor compression thus low percentage compression ratio. Therefore, the nature of the data that represented 2367KB could have affected the performance of the compression algorithm resulting to the low percentage compression.

This research showed that larger files recorded a higher processing time than smaller files. This is a unique characteristic comes in when you look at the change in processing time against time. Smaller files showed the change was lower while the files showed the change was larger. This could be attributed by the rate at which the lookup table grew.

In this research, large file sizes demonstrated a slightly lower percentage compression ratio compared to small file sizes. The difference between the best and the lowest percentage compression ratio is approximately 2.4251%. Mark et al. (1995) argues when large files are compressed, the compression program collects more statistics which take up more space. This causes large files to exhibit relatively low percentage compression ratios.

However, Dhiah et al. (2010), argued that losseless compression algorithms techniques create lookup tables for mapping symbols to binary codes during compression process. Dhiah et al. (2010) further said that, lookup tables in large documents consume small space in comparison to the encoded symbols (binary codes) while lookup documents in small documents could be larger than the encoded data. This resulted in large data having high compression ratio compared to smaller documents. Dhiah et al. (2010), was actually concerned with small file sizes.

Since this research worked on relatively large file sizes, we can go with Mark et al. (1995) argument and say that Gzip compression collected more statistics for large file sizes which made them protray larger percentage compression ratios as compared to smaller file sizes. Microsoft has implemented Gzip compression algorithm in its IIS from Windows Server 2000 (Evers,

2014) to the current Windows 8 (Microsoft, 2014). Notwithsdanding, further tests can be done in the latest IIS platforms to establish its percentage compression ratio and performance.

Compression is important in improving performance in SOAP communication because it improves bandwidth utilization and response time (Seyyed et al., 2011) (Ivan et al., 2008) (Chandra et al., 2012) (Alex et al., 2008) (Tanakorn et al., 2008) (Tomasz et al., 2010). This is essential in environments that face poor bandwith. This research simulated a disadvantaged network environment by limiting bandwith to 10mbps. In as much as compression improves bandwidth utilization and response time, it has its tradeoffs which include extra processing time and CPU utilization. Some researchers like Kiran et al. (2003) argued that these tradeoffs were not benefitial. However, since the increase in hardware capabilities, these tradeoffs are beneficial and not as costly as increasing bandwidth which is widely under constrains (Ivan et al., 2008). We further recommend that for optimal performance when implemeting compression, we need to employ hardware with powerful processors and RAM.

We also need to be careful when applying compression. Tomasz et al. (2010) and Alex et al. (2008) ascertained that lossless compression will not work for high entropy (highly disordered) data e.g. already compressed data, random data or encrypted data as it results in expansion rather than compression.

We can therefore say that compression is an important technique in improving bandwidth utilization and transfer time. Transfer time is further discussed in section 4.7.2.

4.7.2 DISCUSIONS OF RESULTS FOR TIME TO TRANSFER SOAP MESSAGE

Figure 4.4 illustrates a graph of change in transfer time against file size of compressed hybrid architecture and uncompressed hybrid architecture SOAP requests. The lines representing compressed and uncompressed hybrid architecture rose steadily indicating that smaller files exhibited a smaller transfer time compared to the large file. Moreover, the line representing hybrid compressed run slightly below hybrid uncompressed. This showed that the hybrid compressed recorded a lower transfer time hybrid compared to hybrid uncompressed. Nevertheless, Figure 4.5 shows that smaller files displayed a higher percentage improvement time than larger files. File size 387KB recorded the lowest percentage improvement.

The S.I unit of measuring data in a communication channel is Kilobyte (KB). Speed of data in a communication channel is measure in megabytes per second (mbps) i.e. 1,000,000 bytes in 1 second (Andrew et al., 2011). File sizes in this research are measured in kilobytes (KB). We configured the network bandwidth speed to a constant of 10mbps in this research's experiment set up in section 4.1. Looking at the formulae of calculating bandwidth speed equation (4.23) (Andrew et al., 2011), we can derive time using equation (4.24). Meaning if we have a large file size we expect time to increase. This explains why large files exhibited higher transfer times.

$$\text{Bandwidth speed (mbps)} = \text{Megabytes (MB)} \div \text{Time(sec)} \tag{4.23}$$

$$\text{Time (sec)} = \text{Megabytes (MB)} \div \text{Bandwidth (mbps)} \tag{4.24}$$

After compression, hybrid compressed reduced in size, having an average compression ratio percentage of 67.01% as shown in Table 4.15. File size after compression with Gzip algorithm, exhibited a reduced transfer time of an average improvement percentage of 74.87%, as show in Table 4.12. Similarly, several researchers noted that large files exhibited more transfer time compared to smaller files. Compression improved transfer time of these large files (Ivan et al., 2008), (Tomasz et al., 2010), (Tanakorn et al., 2008), (Alex et al., 2008) (Kiran et al., 2003).

However, transfer time can be affected by traffic in a network. This traffic could be caused by other applications utilizing the communication channel. This could have caused the poor percentage improvement of file size 387KB in Figure 4.5. When conducting such experiments, it is advisable to isolate the network or set up a LAN free from interference of other computers or internet. At times it is hard to keep this ideal environment constant because even the default programs in the computer generate traffic unpredictably. In spite of these, we tried to mitigate them by collecting three samples and doing an average of the results as show in this research's data collection procedures in section 4.2. This was to reduce bias while improving accuracy.

Nothwithstanding, Hosein et al. (2012) noted that when using a compression algorithm, if the data is more than a specified threshold it may increase the response time, otherwise it will degrade the perfromance. Therefore, when choosing a compression algorithm, we need to do a thorough review including their compression limits. With this knowledge, a web servers can be configure with this limit e.g. Microsoft's IIS (Evers, 2014) can be configured to compres data that is larger than than a cetain file size.

When we compared this research's hybrid architecture compressed results with Seyyed et al. (2011) results, in Figure 4.6, Figure 4.6, and Table 4.19, we see that compressed hybrid architecture showed a better performance. This could be attributed by hybrid compressed having a better compression ratio which resulted to a better transfer time in the communication channel.

Computer users anticipate faster services when running application. Transfer time of SOAP messages improve speed of information flowing in a communication channel. This research supports compression in improving SOAP performance as supported by Ivan et al. (2008), Tomasz et al. (2010), Tanakorn et al. (2008), Alex et al. (2008) and Kiran, et al. (2003).

4.7.3 DISCUSIONS OF RESULTS FOR TIME TO PROCESS REQUEST

Figure 4.8 represented a graph of time to process SOAP request against change in file size of compressed and uncompressed hybrid architectures. The trend shows that both lines rose moderately. The line representing hybrid uncompressed ran below hybrid compressed. Moreover, Figure 4.9 represented the percentage degraded time to process SOAP request against change in file size of compressed and uncompressed hybrid architectures. The line representing the percentage degradation rises steadily then drops slightly. This research's hybrid compressed experiences an average percentage degradation of 45.53% as shown in Table 4.13.

Experiments in this research deduce that hybrid compressed took more time when being processed on the server-side. This is due to the fact that compression has a tradeoff which is extra CPU processing time (Kiran et al., 2003), (Ivan et al., 2008), (Hosein et al., 2012) (Chandra et al., 2012) (Alex et al., 2008) (Tanakorn et al., 2008) (Tomasz et al., 2010). This overhead causes the web server to spend an extra time in compressing data causing compressed hybrid architecture to experience more processing time of its request.

The percentage degraded time in processing SOAP request in Figure 4.9 showed that smaller files had a low degradation percentage. Meaning smaller files are less affected with the CPU processing time tradeoff. This could be because smaller files require less CPU processing time when mapping its actual data to look up tables and encoded data. Actually, large files have a lot of bytes compared to smaller files, which need to go through the compression program for mapping to look up tables and encoded data (Mark et al., 1995).

When we compared this research's compressed hybrid architecture with Seyyed et al. (2011) as shown in Figure 4.8 and Figure 4.9. We see that compressed hybrid architecture generally recorded an average percentage improvement of 42.01% as show in Table 4.20. Different compression algorithms require different CPU processing time (Dhiah et al., 2010). This research's compressed hybrid architecture done with Gzip compression algorithm required relatively low processing time compared to Seyyed et al. (2011).

Moreover, its noted that we need to be monitoring the percentage processor time counter of our computers. Compression works best if the percentage processor time counter is below 80% and there is adequate disk space, otherwise, the time to process request will be degraded. Additionally, enabling dynamic content in Internet Information Services (IIS) while generating large volumes of dynamic content on the server will impact negatively the time taken to process compressed requests as it results to high processor usage (Evers, 2014). Past these limits the CPU takes excessive time to compress files. All these were put into consideration while setting this research's experiments.

Compression is preferred for disadvantaged networks (Tomasz et al., 2010) (Tanakorn et al., 2008) (Alex et al., 2008). Compression degrades time to process requests at the server. However, the server hardware can be upgrade to mitigate this poor performance. Looking at Moores's Law which describes the doubling of transistors in integrated circuits (IC) in computer hardware in a span of approximately two years (Intel Corporation, 2014). From this point of view and Ivan et al. (2008) remarks, in future we anticipate improved hardware capabilities that will handle compression processing. As part of recommendation for future work, we recommed a review of a compression algorithm whose perfromance tradeoffs will not affect negitively time to process SOAP request at the server-side and RTT in general.

4.7.4 DISCUSIONS OF RESULTS FOR ROUND TRIP TIME (RTT)

Figure 4.12 and Figure 4.13 illustrate a line graph of RTT against file size of compressed and uncompressed hybrid architectures. We generally deduce a degraded performance in RTT. Table 4.16 shows an average degraded performance of 32.91% in RTT.

RTT is the time required for a SOAP message to traverse a network. RTT is the sum of time to process and time transfer SOAP request as shown in equation (4.12). In this research, Figure 4.12 and Figure 4.13, RTT was majorly affected by time to process SOAP requests as discussed in section 4.7.3. Despite the fact that time to transfer and bandwidth utilization was improve with the introduction of Gzip compression, time to process request was affected negatively which affected RTT generally when we compare compressed and uncompressed hybrid architectures. Time to transfer SOAP messages and time to process SOAP requests were discussed in section 4.7.2 and 4.7.3 respectively.

Nevertheless, Figure 4.14, Figure 4.15, and Table 4.21 compare this research’s compressed hybrid architecture and Seyyed et al. (2011) performance. We found out that there was a general improved performance having an average percentage RTT improvement of 44.04%. This was attributed by improving both transfer and process time of SOAP messages. This was discusses in section 4.7.2 and section 4.7.3.

4.7.5 DISCUSIONS OF RESULTS FOR THROUGHPUT

We considered equation (4.15) and equation (4.16) to calculate throughput. We deduced throughput from average RTT. We can presume that throughput is the reciprocal of RTT. This might not be true if we consider latency i.e. time a request wasted in waiting in queue to be serviced/ processed. Latency must be considered while evaluating throughput (Andrew et al., 2011). It should be added to RTT while evaluating throughput. Nevertheless, we carefully did not consider latency while running our experiments. We ensured that one experimental request run to completion before running another one as discussed in this research's experiments setup in section 4.1. This avoided queuing of request at the server thus avoiding latency that could be difficult to measure. Out of these, see how RTT affects throughput.

Throughput = Request ÷ RTT

Therefore for 1 request, Throughput = 1 ÷ RTT

OR

Throughput = Reciprocal of RTT

Considering latency, Throughput = 1 ÷ (RTT + Latency)

Figure 4.16 and Table 4.22 show aggregated throughput results of compressed hybrid architecture, uncompressed hybrid architecture and Seyyed et al. (2011). It was noted in section 4.6.5 that hybrid compressed recorded lower throughput values than hybrid uncompressed. There are many ways of evaluating throughput, in this research we evaluated throughput as the number of requests that can be processed by a web server per second (Tekli et al., 2011).

From the discussion in section 4.7.4, we saw that hybrid compressed recorded lower RTT compared to hybrid uncompressed resulting to hybrid uncompressed exhibiting better throughput values than hybrid compressed. Likewise, Seyyed et al. (2011) registered lower throughput values compared to hybrid compressed because Seyyed et al. (2011) recorded higher RTT values than hybrid compressed. This resulted to hybrid compressed recording better throughput values.

Throughput is significant because people want servers with high throughput values to support more demanding uses (Andrew et al., 2011). Throughput can be improved by improving RTT of requests. Improvements procedures were pointed out in section 4.7.3 when we were discussing time to process requests at the server-side. Time to process requests at the server-side degrade RTT.

CHAPTER FIVE

SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

5.1 SUMMARY AND CONCLUSION

This research has highlighted web services and its implementation in service-oriented architecture. A web service extends the World Wide Web infrastructure to provide the means for software to connect to other software applications. Web services integrate heterogeneous software applications through standard web protocols and data formats e.g. HTTP, XML and SOAP.

Simple Object Access Protocol (SOAP) is an implementation of web services. SOAP beats its competitors due to its simplicity, flexibility, robustness, extensibility and inter-operability in heterogeneous systems. Some of its rivals are CORBA and Java RMI. SOAP messages structure is composed of a SOAP envelope, header, and body. SOAP transmits its data represented in XML. XML documents are huge in size and verbose which results to high network traffic and XML parsing and processing causes a high computational burden leading to high latency.

This research reviewed some of the techniques employed to improve SOAP performance in communication. Out of this, this research developed a hybrid architecture included: client-side caching, simple server-side database queries, compression technique and document-literal style description of WSDL. The hybrid architecture was implemented in a software and run in an experimental environment where we collected data.

The data was processed and evaluated against SOAP performance metrics: compression ratio percentage, SOAP request transfer time, SOAP request processing time round trip time, and throughput. We found that compression improved compression ratio and SOAP transfer time.

However, compression compromised SOAP processing time, round trip time and throughput. Nevertheless, compression is supported by several researchers in disadvantaged networks i.e. 10mbps. The general impact of this is that, in poor networks software end users will experience a relatively high turnaround time and low throughput. Nevertheless, bandwidth utilization was evident. Better bandwidth utilization saves cost in communication.

5.2 RECOMMENDATIONS FOR FUTURE WORK

In this research, we found that processing SOAP request on the server-side took longer when the file is compressed than when it is uncompressed. An optimization of the compression process in the server-side can greatly improve performance. Moreover, we need to ascertain a better compression algorithm that: has better compression ratio percentage, utilizes minimal CPU resource, and is interoperable among clients e.g. web browsers. Nevertheless, small file sizes exhibited better compression percentages with Gzip algorithm, further research can be done on large files. Furthermore, XML parsing and processing causes a high computational burden leading to high latency. Further research can be invested here.

REFERENCES

1. Aaron S. http://msdn.microsoft.com/en-us/library/ms996486.aspx. [Created: October 2003] [Cited: November 25, 2013]

2. Alex N; Paul G; Shiping C. A Study of the Impact of Compression and Binary Encoding on SOAP Performance, 2008, pp. 46-56.

3. Amer A, Ahmed K. Multimedia Web Services Performance: Analysis and Quantification of Binary Data Compression. Journal of Multimedia, 2011, Vol. VI.

4. Andrew T. and Wetherall D. Computer Networks, Pearson, 2011. Edition 5th.

5. Apache Foundation. Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html. [Cited: November 21, 2013]

6. Ashkan P; Dan M. A Cloud Service for Adaptive Digital Music Streaming. 2012 Eighth International Conference on Signal Image Technology and Internet Based Systems, 2012.

7. Barry & Associates Service-Oriented Architecture (SOA) Definition. Service Architecture. http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html. [Cited: August 13, 2013]

8. Behrouz M; Parinaz S. SOAP Serialization Performance Enhancement DESIGN AND IMPLEMENTATION OF A MIDDLEWARE. (IJCSIS) International Journal of Computer Science and Information Security, Vol.6, No. 1, 2009. pp. 106-110.

9. Bianco P, Rick K and Paulo M. Evaluating a Service-Oriented Architecture, Hanscom, Carnegie Mellon University, 2007.

10. Bob K. Advanced Computer Networks. http://web.cs.wpi.edu/~rek/Nets2/C10/C10.html. [Cited: November 2013, 2013]

11. Booth D; Hugo H. Francis M; Eric N; Michael C; Chris F; David O. Web Services Architecture http://www.w3.org/TR/ws-arch/#whatis. [Crated: February 11, 2004] [Cited: August 13, 2013]

12. Chandra M; Rajendra C. Caching and SOAP compression techniques in Service Oriented Architecture. International Journal of Advanced Research in Computer Engineering & Technology, 2012, Vol. I, ISSN: 2278–1323.

13. Chiu K; Govindaraju M; Bramley R. Investigating the Limits of SOAP Performance for Scientific Computing. Proceedings of 11th. IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02), Edinburgh, IEEE, 2002, - pp. 246-254.

14. CISCO. http://www.cisco.com/en/US/products/ps6558/products_ios_technology_home.html. [Cited: November 21, 2013]

15. Constantin R. HTTP Compression. http://www.http-compression.com/. [Created: August 12, 2011] [Cited: March 29, 2014]

16. Coulouris G, Jean D and Kindberg T. Distributed Systems Concepts and Design, Essex, Pearson Education Limited, 2009, Vol. IV, pp. 789.

17. Davis D; Parashar M. Latency Performance of SOAP Implementations. Proceedings of IEEE Cluster Computing and the GRID 2002 (CCGRID'02, Berlin, IEEE, 2002.

18. Dhiah A; Ibrahim K. SOAP Web Services Compression using Variable and Fixed Length Coding. 2010 Network Computing and Applications, 2010.

19. Don B; David E; Gopal K; Andrew L; Henrik F; Dave W. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/. [Created: May 08, 2000] [Cited: June 06, 2013]

20. Endrei M; Jenny A; Ali A; Sook C; Philippe C; Pal K; Luo M; Tony N. Patterns: Service Oriented Architecture and Web Services, RedBooks, 2004, Vol. I.

21. Erik C; Francisco C; Greg M; Sanjiva W. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl. [Created: March 15, 2001] [Cited: November 25, 2013]

22. Evers D. Using HTTP Compression On Your IIS 5.0 Web Site
[Online] http://technet.microsoft.com/en-us/library/bb742379.aspx. [Created: 2014] [Cited: April 10, 2014]

23. Fabian B; Greg E; Karsten S; Patrick W. Efficient Wire Formats for High Performance Computing, Georgia, IEEE, 2000.

24. Fiddler. http://fiddler.wikidot.com/timers. [Cited: April 2014]

25. Girish T; Mudholkar R; Jadhav B. JAX-WS Web Service for Transferring Image. International Journal on Computer Science and Engineering (IJCSE), Science Publications, 2013, pp. 63-69.

26. Grosso W. Java RMI, O'Reilly, 2001.

27. Guillermo G. How to enable gzip compression in Apache 2.x. http://www.garron.me/en/linux/enable-gzip-mod_deflate-compression-apache.html. [Created: August 24, 2011] [Cited: March 29, 2014]

28. Hazem E; Nikos M. Studying the Efficiency of XML Web Services for Real-Time Applications.

29. Hosein S; Hassan R; Hajar H. The Effects of Data Compression on Performance of Service-Oriented Architecture (SOA). International Journal of Emerging Trends & Technology in Computer Science (IJETTCS), 2012, Vol. I, ISSN 2278-6856.

30. Hou Z; ZHAI H; Gao G. A Study on Web Services Performance. Third International Symposium on Electronic Commerce and Security Workshops, 2010, ISBN: 978-952-5726-11-4.

31. IBM. New to SOA and web services. http://www.ibm.com/developerworks/webservices/newto/index.html#ibm-pcon. [Cited: August 13, 2013]

32. IBM. The structure of a SOAP message [Online] . - http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp?topic=%2Fcom.ibm.cics.ts31.doc%2Fdfhws%2Fconcepts%2Fsoap%2Fdfhws_message.htm. [Cited: August 16, 2013]

33. IBM. Web services overview. http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=%2Fcom.ibm.etools.webservice.doc%2Fconcepts%2Fcws.html. [Cited: August 13, 2013]

34. IBM. Web services performance best practices. http://www14.software.ibm.com/webapp/wsbroker/redirect?version=compass&product=was-express-dist&topic=rwbs_perfbestpractices.[Created: July 11, 2013]. [Cited: August 06, 2013]

35. Intel Corporation. Moore's Law and Intel Innovation. http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html. [Cited: April 25, 2014]

36. IONA Technologies. CORBA Programmer's Guide, Java. Sun Microsystems, Inc, 2005. - Vol. 6.3.

37. Ivan I, Saso G; Dimitar T. Enhancing Performance of Web Services in Mobile Applications by SOAP Compression, 2008.

38. JMeter Apache. Apache Software Foundation. http://jmeter.apache.org/. [Cited: November 21, 2013]

39. Junichi T; Oliver P; Arsany S. WReX: A Scalable Middleware Architecture to Enable XML Caching for Web-Services. International Federation for Information Processing, 2005.

40. Kho P. Enhanced SOAP Performance for Low Bandwidth Environments , 2007.

41. Kiran D; Daniel A. SOAP Optimization via Client-side Caching, Manhattan, Citeseer, 2003.

42. Kohlhoff C; Steele R. Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems, World Wide Web (WWW), 2003.

43. Linthicum D. Service Oriented Architecture (SOA). http://msdn.microsoft.com/en-us/library/bb833022.aspx. [Cited: August 13, 2013]

44. Liquid Technologies, Product : Liquid XML Compression Library. http://www.liquid-technologies.com/Liquid-Products/XMLCompression/XMLCompressionExample.aspx. [Cited, August 13, 2013]

45. Mark E; Jenny A; Ali A; Sook C; Philippe C; Pal K; Luo M; Tony N. Patterns: Service Oriented Architecture and Web Services, RedBooks, 2004, Vol. I.

46. Mark N. and Jean-loup G. The Data Compression Book, 1995. Edition 2nd.

47. Martinez E. G. Performance Analysis of Web Applications, 2005.

48. Masoud K. A Look at WSDL 2.0. http://soa.dzone.com/news/look-wsdl-20. [Created: April 23, 2009]. [December 24, 2013]

49. Matthieu S; Jens K; Vadym B; Alexander Z; Hasso P. Data Loading & Caching Strategies in Service-Oriented Enterprise Applications. Congress on Services, 2009. ISSN 978-0-7695-3708-5/09.

50. Microsoft Download Center. Microsoft Network Monitor 3.4. http://www.microsoft.com/en-us/download/details.aspx?id=4865. [Cited: August 05, 2013]

51. Microsoft. Enable HTTP Compression of Dynamic Content (IIS 7). http://technet.microsoft.com/en-us/library/cc753681%28v=ws.10%29.aspx. [Cited: April 25, 2014]

52. Microsoft. COM: Component Object Model Technologies. http://www.microsoft.com/com/default.mspx. [Cited: August 23, 2013]

53. Microsoft. Using HTTP Compression for Faster Downloads (IIS 6.0). http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/25d2170b-09c0-45fd-8da4-898cf9a7d568.mspx?mfr=true. [Cited: March 29, 2014]

54. Microsoft. Web Services. http://msdn.microsoft.com/en-us/library/ms950421.aspx. [Cited: August 14, 2013]

55. Mostafa N. Software Performance Profiling, ACM, 2008.

56. Nayef A; Michael L. Differential Checkpointing for Reducing Memory Requirements in Optimized SOAP Deserialization. Grid ComputingWorkshop, IEEE, New York, 2005. ISSN 0-7803-9493-3/05.

57. Nayef A; Michael L. Differential Deserialization for Optimized SOAP Performance. ACM/IEEE conference on Supercomputing, Binghamton, IEEE, 2005, pp. 1-12.

58. Nayef A; Michael L. Lightweight Checkpointing for Faster SOAP Deserialization. International Conference on Web Services, New York, 2006. ISSN 0-7695-2669-1/06.

59. Nayef A; Michael L; Madhusudhan G. Differential Serialization for Optimized SOAP Performance. International Symposium on High Performance Distributed Computing (HPDC), New York , 2004, pp. 55-64.

60. NEOTYS. NeoLoad Cloud Testing. http://www.neotys.com/introduction/neoload-cloud-testing.html. [Cited: November 21, 2013]

61. Papazoglou M. Web Services: Principles and Technology, Pearson Education Limited, 2008, Vol. I.

62. Pressman R. Software Engineering A Practitioner's Approach, McGraw-Hill High Education, 2010.

63. Qusay H. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI). http://www.oracle.com/technetwork/articles/javase/soa-142870.html. [Cited: August 13, 2013]

64. Sencha. Custom Grid Filters Example (local filtering). www.sencha.com. [Cited: December 27, 2013]

65. Seyyed H; Roya Z. A Combination Approach for Improvement Web Service Performance. Proceeding of the International MultiConference of Engineers and Computer Scientists. International Assiciation of Engineers, Hong Kong, 2011, Vol I.

66. Shariq H; Zhaoshun W; Ibrahima K; Diop A. Web Service Testing Tools: A Comparative Study, 2012.

67. SmartBear. SoapUI. http://www.soapui.org/. [Cited: November 21, 2013]

68. Snell J; Doug T. Pavel K. Programming Web Services with SOAP. O'Reilly, Canada, 2001, Vol. I, pp. 6-7.

69. Sommerville I. Software Engineering, Pearson Education, 2011, Edition 9th.

70. Storm. http://storm.codeplex.com/. [Cited: November 21, 2013].

71. Suzumura T; Tatsubori M. Optimizing Web Services Performance by Differential Deserialization . IEEE International Conference on Web Services, NY, 2005, pp. 185- 192.

72. Takase T; Miyashita H; Tatsubori M; Suzumura T. An Adaptative, Fast and Safe XML Parser Based on Byte Sequence Memorization. World Wide Web (WWW) Conference. World Wide Web (WWW), 2005, pp. 692 - 701.

73. Tanakorn W; Kitti K; Chuleerat J. A Simple Approach to Optimized Text Compression's Performance. 4th International Conference on Next Generation Web Services Practices, 2008.

74. Tekli J; Ernesto D; Richard C; Gabriele G. Similarity-based SOAP Processing Performance and Enhancement. IEEE, 2011, Vol. 5, pp. 387-403.

75. Terelik. HTTP/HTTPS traffic recording http://fiddler2.com/features. [Cited:November 21, 2013]

76. Terrence D. Tools and Techniques for .NET Code Profiling. http://msdn.microsoft.com/en-us/magazine/hh288073.aspx. [Cited: November 21, 2013]

77. Tomasz P; Joanna S; Marek A. Efficiency of compression techniques in SOAP, 2010, pp. 199-211.

78. Wireshark. http://www.wireshark.org. [Cited: March 29, 2014]

APPENDICES

APPENDIX A: RAW UNCOMPRESSED HTTP SOAP TRAFFIC

Table A-1: COMP A - raw uncompressed HTTP SOAP traffic hybrid architecture

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500**	**501**	**501**	**501**	**501**
Sent: Body (KB)	**0**	**0**	**0**	**0**	**0**
Received: Header (KB)	**223**	**223**	**224**	**224**	**224**
Received: Body (KB)	**199,777**	**386,777**	**2,366,776**	**5,527,776**	**7,299,776**
ClientConnected: (Time)	**7:12:34.609**	**7:12:34.609**	**7:12:34.609**	**7:12:34.609**	**7:12:34.609**
ClientBeginRequest: (Time)	**7:13:05.046**	**7:13:05.046**	**7:13:27.265**	**7:13:50.890**	**7:14:12.109**
GotRequestHeaders: (Time)	**7:13:05.046**	**7:13:05.046**	**7:13:27.265**	**7:13:50.890**	**7:14:12.109**
ClientDoneRequest: (Time)	**7:13:05.046**	**7:13:05.046**	**7:13:27.265**	**7:13:50.890**	**7:14:12.109**
Determine Gateway: (Time)	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
DNS Lookup:	**0ms**		**0ms**	**0ms**	**0ms**
TCP/IP Connect:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
HTTPS Handshake:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
Time ServerConnected: (Time)	**7:13:05.046**	**7:13:05.046**	**7:13:05.046**	**7:13:05.046**	**7:13:05.046**
FiddlerBeginRequest: (Time)	**7:13:05.046**	**7:13:05.046**	**7:13:27.265**	**7:13:50.890**	**7:14:12.109**
ServerGotRequest: (Time)	**7:13:05.046**	**7:13:05.046**	**7:13:27.265**	**7:13:50.890**	**7:14:12.109**
ServerBeginResponse: (Time)	**7:13:05.093**	**7:13:05.093**	**7:13:27.546**	**7:13:51.531**	**7:14:12.937**
GotResponseHeaders: (Time)	**7:13:05.093**	**7:13:05.093**	**7:13:27.546**	**7:13:51.531**	**7:14:12.937**
ServerDoneResponse: (Time)	**7:13:05.109**	**7:13:05.109**	**7:13:27.578**	**7:13:51.593**	**7:14:13.000**
ClientBeginResponse: (Time)	**7:13:05.109**	**7:13:05.109**	**7:13:27.578**	**7:13:51.593**	**7:14:13.015**
ClientDoneResponse: (Time)	**7:13:05.109**	**7:13:05.109**	**7:13:27.578**	**7:13:51.609**	**7:14:13.031**

Table A-2: COMP B - raw uncompressed HTTP SOAP traffic

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body (KB)	-	-	-	-	-
Received: Header (KB)	223	223	224	224	224
Received: Body (KB)	199,777	386,777	2,366,776	5,527,776	7,299,776
ClientConnected: (Time)	7:10:00.062	7:11:05.140	7:11:05.140	7:11:05.140	7:11:05.140
ClientBeginRequest: (Time)	7:10:18.859	7:11:05.156	7:11:29.250	7:11:51.375	7:12:08.093
GotRequestHeaders: (Time)	7:10:18.859	7:11:05.156	7:11:29.250	7:11:51.375	7:12:08.093
ClientDoneRequest: (Time)	7:10:18.859	7:11:05.156	7:11:29.250	7:11:51.375	7:12:08.093
Determine Gateway: (Time)	0ms	0ms	0ms	0ms	0ms
DNS Lookup:	0ms	0ms	0ms	0ms	0ms
TCP/IP Connect:	0ms	0ms	0ms	0ms	0ms
HTTPS Handshake:	0ms	0ms	0ms	0ms	0ms
Time ServerConnected: (Time)	7:10:10.765	7:10:10.765	7:10:10.765	7:10:10.765	7:10:10.765
FiddlerBeginRequest: (Time)	7:10:18.859	7:11:05.156	7:11:29.250	7:11:51.375	7:12:08.093
ServerGotRequest: (Time)	7:10:18.859	7:11:05.156	7:11:29.250	7:11:51.375	7:12:08.093
ServerBeginResponse: (Time)	7:10:18.890	7:11:05.203	7:11:29.515	7:11:52.015	7:12:08.906
GotResponseHeaders: (Time)	7:10:18.890	7:11:05.203	7:11:29.515	7:11:52.015	7:12:08.906
ServerDoneResponse: (Time)	7:10:18.890	7:11:05.203	7:11:29.531	7:11:52.046	7:12:08.968
ClientBeginResponse: (Time)	7:10:18.890	7:11:05.203	7:11:29.531	7:11:52.062	7:12:08.968
ClientDoneResponse: (Time)	7:10:18.890	7:11:05.203	7:11:29.546	7:11:52.062	7:12:08.984

Table A-3: COMP C - raw uncompressed HTTP SOAP traffic

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500**	**501**	**501**	**501**	**501**
Sent: Body (KB)	**0**	**0**	**0**	**0**	**0**
Received: Header (KB)	**223**	**223**	**224**	**224**	**224**
Received: Body (KB)	**199,777**	**386,777**	**2,366,776**	**5,527,776**	**7,299,776**
ClientConnected: (Time)	**18:51:36.125**	**18:53:05.781**	**18:53:05.781**	**18:53:05.781**	**18:53:05.781**
ClientBeginRequest: (Time)	**18:52:07.828**	**18:53:05.781**	**18:53:24.265**	**18:53:47.531**	**18:54:10.062**
GotRequestHeaders: (Time)	**18:52:07.828**	**18:53:05.781**	**18:53:24.265**	**18:53:47.531**	**18:54:10.062**
ClientDoneRequest: (Time)	**18:52:07.828**	**18:53:05.781**	**18:53:24.265**	**18:53:47.531**	**18:54:10.062**
Determine Gateway: (Time)	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
DNS Lookup:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
TCP/IP Connect:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
HTTPS Handshake:	**0ms**	**0ms**	**0ms**	**0ms**	**0ms**
Time ServerConnected: (Time)	**18:51:36.140**	**18:51:36.140**	**18:51:36.140**	**18:51:36.140**	**18:51:36.140**
FiddlerBeginRequest: (Time)	**18:52:07.828**	**18:53:05.781**	**18:53:24.265**	**18:53:47.531**	**18:54:10.062**
ServerGotRequest: (Time)	**18:52:07.828**	**18:53:05.781**	**18:53:24.265**	**18:53:47.531**	**18:54:10.062**
ServerBeginResponse: (Time)	**18:52:07.859**	**18:53:05.843**	**18:53:24.546**	**18:53:48.156**	**18:54:10.906**
GotResponseHeaders: (Time)	**18:52:07.859**	**18:53:05.843**	**18:53:24.546**	**18:53:48.156**	**18:54:10.906**
ServerDoneResponse: (Time)	**18:52:07.859**	**18:53:05.859**	**18:53:24.578**	**18:53:48.218**	**18:54:10.984**
ClientBeginResponse: (Time)	**18:52:07.859**	**18:53:05.859**	**18:53:24.593**	**18:53:48.234**	**18:54:11.000**
ClientDoneResponse: (Time)	**18:52:07.859**	**18:53:05.859**	**18:53:24.593**	**18:53:48.234**	**18:54:11.015**

APPENDIX B: RAW UNCOMPRESSED PRE-PROCESSED RESULTS

Table B-1: COMP – A, uncompressed hybrid achitecture pre-processed results

File size (KB)	200	387	2367	5528	7300
Sent: Header (KB)	500	501	501	501	501
Sent: Body(KB)	-	-	-	-	-
Total Request Header (KB)	500	501	501	501	501
Received: Header (KB)	223	223	224	224	224
Received: Body (KB)	199,777	386,777	2,366,776	5,527,776	7,299,776
Total Response Header (KB)	200,000	387,000	2,367,000	5,528,000	7,300,000
ClientConnected: (ms)	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000	0:00:00.000
ClientBeginRequest: (ms)	0:00:30.437	0:00:30.437	0:00:52.656	0:01:16.281	0:01:37.500
GotRequestHeaders: (ms)	0:00:30.437	0:00:30.437	0:00:52.656	0:01:16.281	0:01:37.500
ClientDoneRequest: (ms)	0:00:30.437	0:00:30.437	0:00:52.656	0:01:16.281	0:01:37.500
FiddlerBeginRequest: (ms)	0:00:30.437	0:00:30.437	0:00:52.656	0:01:16.281	0:01:37.500
ServerGotRequest: (ms)	0:00:30.437	0:00:30.437	0:00:52.656	0:01:16.281	0:01:37.500
ServerBeginResponse: (ms)	0:00:30.484	0:00:30.484	0:00:52.937	0:01:16.922	0:01:38.328
GotResponseHeaders: (ms)	0:00:30.484	0:00:30.484	0:00:52.937	0:01:16.922	0:01:38.328
ServerDoneResponse: (ms)	0:00:30.500	0:00:30.500	0:00:52.969	0:01:16.984	0:01:38.391
ClientBeginResponse: (ms)	0:00:30.500	0:00:30.500	0:00:52.969	0:01:16.984	0:01:38.406
ClientDoneResponse: (ms)	0:00:30.500	0:00:30.500	0:00:52.969	0:01:17.000	0:01:38.422

Table B-2: COMP – B, uncompressed hybrid architecture pre-processed results

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500**	**501**	**501**	**501**	**501**
Sent: Body(KB)	**-**	**-**	**-**	**-**	**-**
Total Request Header (KB)	**500**	**501**	**501**	**501**	**501**
Received: Header (KB)	**223**	**223**	**224**	**224**	**224**
Received: Body (KB)	**199,777**	**386,777**	**2,366,776**	**5,527,776**	**7,299,776**
Total Response Header (KB)	**200,000**	**387,000**	**2,367,000**	**5,528,000**	**7,300,000**
ClientConnected: (ms)	**0:00:00.000**	**0:00:00.000**	**0:00:00.000**	**0:00:00.000**	**0:00:00.000**
ClientBeginRequest: (ms)	**0:00:18.797**	**0:00:00.016**	**0:00:24.110**	**0:00:46.235**	**0:01:02.953**
GotRequestHeaders: (ms)	**0:00:18.797**	**0:00:00.016**	**0:00:24.110**	**0:00:46.235**	**0:01:02.953**
ClientDoneRequest: (ms)	**0:00:18.797**	**0:00:00.016**	**0:00:24.110**	**0:00:46.235**	**0:01:02.953**
FiddlerBeginRequest: (ms)	**0:00:18.797**	**0:00:00.016**	**0:00:24.110**	**0:00:46.235**	**0:01:02.953**
ServerGotRequest: (ms)	**0:00:18.797**	**0:00:00.016**	**0:00:24.110**	**0:00:46.235**	**0:01:02.953**
ServerBeginResponse: (ms)	**0:00:18.828**	**0:00:00.063**	**0:00:24.375**	**0:00:46.875**	**0:01:03.766**
GotResponseHeaders: (ms)	**0:00:18.828**	**0:00:00.063**	**0:00:24.375**	**0:00:46.875**	**0:01:03.766**
ServerDoneResponse: (ms)	**0:00:18.828**	**0:00:00.063**	**0:00:24.391**	**0:00:46.906**	**0:01:03.828**
ClientBeginResponse: (ms)	**0:00:18.828**	**0:00:00.063**	**0:00:24.391**	**0:00:46.922**	**0:01:03.828**
ClientDoneResponse: (ms)	**0:00:18.828**	**0:00:00.063**	**0:00:24.406**	**0:00:46.922**	**0:01:03.844**

Table B-3: COMP – C, uncompressed hybrid architecture pre-processed results

File size (KB)	**200**	**387**	**2367**	**5528**	**7300**
Sent: Header (KB)	**500**	**501**	**501**	**501**	**501**
Sent: Body(KB)	**-**	**-**	**-**	**-**	**-**
Total Request Header (KB)	**500**	**501**	**501**	**501**	**501**
Received: Header (KB)	**223**	**223**	**224**	**224**	**224**
Received: Body (KB)	**199,777**	**386,777**	**2,366,776**	**5,527,776**	**7,299,776**
Total Response Header (KB)	**200,000**	**387,000**	**2,367,000**	**5,528,000**	**7,300,000**
ClientConnected: (ms)	**0:00:00.000**	**0:00:00.000**	**0:00:00.000**	**0:00:00.000**	**0:00:00.000**
ClientBeginRequest: (ms)	**0:00:31.703**	**0:00:00.000**	**0:00:18.484**	**0:00:41.750**	**0:01:04.281**
GotRequestHeaders: (ms)	**0:00:31.703**	**0:00:00.000**	**0:00:18.484**	**0:00:41.750**	**0:01:04.281**
ClientDoneRequest: (ms)	**0:00:31.703**	**0:00:00.000**	**0:00:18.484**	**0:00:41.750**	**0:01:04.281**
FiddlerBeginRequest: (ms)	**0:00:31.703**	**0:00:00.000**	**0:00:18.484**	**0:00:41.750**	**0:01:04.281**
ServerGotRequest: (ms)	**0:00:31.703**	**0:00:00.000**	**0:00:18.484**	**0:00:41.750**	**0:01:04.281**
ServerBeginResponse: (ms)	**0:00:31.734**	**0:00:00.062**	**0:00:18.765**	**0:00:42.375**	**0:01:05.125**
GotResponseHeaders: (ms)	**0:00:31.734**	**0:00:00.062**	**0:00:18.765**	**0:00:42.375**	**0:01:05.125**
ServerDoneResponse: (ms)	**0:00:31.734**	**0:00:00.078**	**0:00:18.797**	**0:00:42.437**	**0:01:05.203**
ClientBeginResponse: (ms)	**0:00:31.734**	**0:00:00.078**	**0:00:18.812**	**0:00:42.453**	**0:01:05.219**
ClientDoneResponse: (ms)	**0:00:31.734**	**0:00:00.078**	**0:00:18.812**	**0:00:42.453**	**0:01:05.234**