Investigation and Development of Quality of Service Management for Web based Services

Managing Quality of Service from
the End-User Perspective

Eamon Dalton

A dissertation submitted to the University of Dublin in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

2000

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Eamon Dalton

September 2000

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Eamon Dalton

September 2000

Acknowledgments

I would especially like to thank my supervisor, Mr Vinny Wade for his guidance in the completion of this dissertation and to Broadcom Eireann for their assistance.

Thanks to my family who have always supported me and to Grainne for her encouragement over the last year.

Summary

With the advent of Web based delivery of business applications and increasing dependency on these services, management of availability and performance are critical. For service providers to offer guarantees on Web based application performance of services delivered via Hypertext Transfer Protocol (HTTP), they must implement Quality of Service (QoS) and have a composite view of service quality across all the components in the service delivery chain including databases, networks and webservers. In most cases this is not possible as different parts of the infrastructure are outsourced to different service providers or managed by the customer. Even with all this information available it is still extremely difficult to diagnose how service performance is perceived by the end user.

This dissertation describes the design and implementation of a framework that supports the monitoring of Web delivered services from the end-users perspective. The service performance and availability can then be compared against a Service Level Agreement.

The framework consists of a number of components. These include a local proxy on the end user's host that intercepts HTTP requests and replies. This allows the monitoring of availability and performance parameters as perceived by the end user. The framework also involved the design of a SLA template specifically for Web delivered services that can map these parameters captured by the proxy in a meaningful way so that it is possible to specify how a service delivered over HTTP should perform from the end-user perspective. The proxy encompasses a number of features including automatic retrieval of a Service Level Agreement (SLA) with a server.

Finally an application was developed that demonstrated how the framework can be applied to monitor SLA compliance for a Web delivered service.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AF	**Assured Forwarding**
ASP	**Application Service Provider**
CRM	**Customer Resource Management**
DNS	**Domain Name Service**
DTD	**Document Type Definition**
EF	**Expedited Forwarding**
Diffserv	**Differentiated Services Framework**
HTTP	**Hypertext Transfer Protocol**
IntServ	**Integrated Services Architecture**
IP	**Internet Protocol**
ISP	**Internet Service Provider**
PHB	**Per-Hop behaviors**
QoS	**Quality of Service**
SLA	**Service Level Agreement**
SLAVSP	**Service Level Agreement Verification Service Provider**
SGML	**Standard Generalized Markup Language**
XML	**Extensible Markup Language**
XSL	**Extensible Stylesheet Language**

1 INTRODUCTION

1.1 Web Delivered Services

As the Internet expands, it presents new ways to provide information to individuals and organisations. Increasingly, the provision of information based services are taking the Application Service Provider (ASP) approach where the service is delivered over the Web and accessed via a web-browser. These Web based services are not just basic web hosting services. They are often complex information management tools that offer data-mining and remote analysis of large volumes of data via the Web with almost all of the content dynamically generated from backend databases. These Application Service Providers are targeting anything from small companies with a couple of employees that lack the skills in house to manage applications locally, to corporations with thousands of employees for which the ASP model offers a more manageable solution to deliver applications to vast numbers of desktops. The revenues from these complex applications are expected to grow rapidly in the next few years, and are estimated to reach $8 billion by 2002 [1].

1.2 Motivation

The big issue is “can Web delivered services be relied on?”. In particular, can performance and availability be comparable to more traditional methods of providing applications and how can this be monitored? ASP delivery has the potential to end up making services more unreliable that ever before due to the best effort nature of the Internet Protocol (IP). “Best effort“ in the context of the Internet Protocol, is where a packet is delivered to its destination as soon as possible but without guarantees on the time taken to get to the destination or even that it will arrive at all. To add to the problem, any Web delivered service is dependent on multiple distributed components that may also implement “best effort” methodologies in how they operate. For some users, “best effort” is simply not good enough particularly when these applications must be depended on for organisation to be able to operate. There is a movement from the democratic system where everyone gets the same level of service, to one where those who are willing to pay more get a better that “best effort” service that is typical

of the Internet. The move from best effort to some guaranteed level of service not only applies to IP traffic, it applies to the components that are being used to deliver services over the Web and even the services themselves. The mechanism that allows guarantees to be implemented is what's termed, "Quality of Service". Quality of Service (QoS) has a number of different meanings depending on the context it is talked about. In the context of networks, it can be define as the ability of a network element (e.g. an application host or router) to have some level of assurance that its traffic and service requirements can be satisfied [2]. In the simplest sense, Quality of Service (QoS) means providing consistent, predictable data delivery service. In other words, satisfying customer application requirements [3]. This latter definition is perhaps, a more suitable general definition, as QoS can be discussed at a number of different architectural levels including network, application and end user QoS. For example, implementing QoS so that specific user(s) can receive a better that "best effort" service from a Webserver has been researched [4].

The formal specification of the QoS guarantees that any of these components implement is defined with what is termed, a Service level Agreement. A Service Level Agreement (SLA) specifies the expected performance and availability of a service component or an overall service that may involve multiple components, or services. However QoS assurances are only as good as the weakest link in the chain between sender and receiver. If any single component to which a service depends on fails, or has performance issues, the entire service is impacted. For example, a Web based Customer Resource Management (CRM) system may be a collection of independent services such as network, Webserver, and database service(s), but from a customer's point of view this is just one service. The entire CRM service may be deemed unavailable if any one of the underlying services (such as the database) is unavailable.

The dependencies of Web delivered services on multiple components makes it apparent that a composite view of QoS and availability for all service components is needed when services are chained and delivered as a single service. However, specific components in a service delivery chain may not managed by the Service Provider and hence this information may not always be available. For example, the network that the end-user is accessing the service from

will likely be managed locally. Even with this information available it is still extremely difficult to correlate QoS of different components and diagnose how a service is performing overall. Extensive research to correlate QoS between service components has been conducted [5-7].

1.2.1 The End-User Perspective

To say that all the components in a Web delivered service were available 100% in a 20 day period has very little relevance as to how a service performs, as perceived by the end-user. Why? not being able to describe end-user perceived performance has been one of the major faults with service management to-date and is even more of a problem now with ASPs. Looking at our CRM example, the end user is interested neither in the QoS guarantees of the Webserver that their browser connects to or the guarantees of the network over which they are accessing a service. End-users view the quality of any Web delivered service in simple terms, **accessibility** and **performance,** as **they** perceive it.

What is also worth remembering is that end user QoS encompasses the failure or success of all other forms of QoS of the components in the service delivery chain and offers a truly meaningful way to determine how a service is performing overall. The fact that the Service Provider typically has no idea of how service performance and availability is perceived, by the user in the scenario described previously is a particular problem. In what other industry in today's market forces will a supplier be content to remain unaware of it's customers perspective on it's products. Were this information available, not alone could it offer the Service Provider a useful diagnostic tool and the customer an ability to specify performance and availability criteria that mean something to them. The ability of Service Providers to offer some level of guarantee to customers on the performance and availability of services from the end user perspective means that it would finally be possible to define meaningful SLAs for Web delivered services. The result - clarity and accountability!

1.3 Objectives

The primary objective of this dissertation is to develop a framework that can support QoS monitoring and Service Level Agreements for Web delivered Services. This has three sections to it.

- To design a client side proxy that supports automatic configuration, monitoring and logging of availability and performance parameters as perceived by the end user of Web delivered services
- The design of a SLA template specifically for Web delivered services that will map these parameters captured by the proxy in a meaningful way so that it is possible to specify how a service delivered over HTTP should perform, from the end-user perspective
- To implement SLA verification and feedback to the Service Provider and Customer so as to notify them of SLA compliance.

1.4 Technical Approach

The completion of the dissertation was undertaken with the following approach.
Initially the area of Quality of Service was examined including QoS in the context of network, applications and service management including composite QoS management Next, the area of Service Level Agreements and the components of a SLA and technologies used to describe SLAs are investigated. The HTTP protocol and it's use as a service delivery mechanism is looked at including what the issues are in using it and the problems it presents for monitoring performance and availability from the end-user perspective. Next the requirements for monitoring service performance and availability from and end-user perspective are addressed including what parameters can be monitored and where they can be captured. A framework was then designed that can describe these requirements and monitor them using a client side proxy to capture the appropriate information. Finally a sample application is built that demonstrates the usefulness of the framework.

1.5 Overview of Dissertation

Chapter 1

Introduction: This chapter presents the motivation for the dissertation, outlines the objectives that the dissertation aims to achieve, the technical approach taken and finally a description of the dissertation structure.

Chapter 2

Background Research: This chapter reviews Quality of Service, Service Level Agreements, Hypertext Transfer Protocol and a number of key technologies used in the dissertation.

Chapter 3

Requirements: Reviews the requirements of the framework including what information needs to be captured, and where the capture process should be implemented. The uses of the captured information are then investigated and finally possible architectures for the framework are examined.

Chapter 4

Design: This chapter describes the design and architecture of the framework that enables the specification and monitoring of SLAs for services delivered via HTTP. The design of the client-side component and information architecture is described.

Chapter 5

Implementation: This chapter describes the implementation of the framework and some of the more complex problems that were encountered in the implementation.

Chapter 6

Evaluation and Conclusion: This chapter evaluates the work completed in the dissertation. It then reviews if the objectives of the dissertation where met and identifies areas for further research.

1.6 Summary

This chapter gives an overview of the move towards the delivery of services over the Web and the introduction of QoS in the components that are used to deliver Web based services. Although QoS can offer specific guarantees on how an individual component in a service delivery chain should perform, for QoS to be effective in service delivery, it needs to be managed in all the components. The problem is that this is not usually possible due to technical difficulties in generating composite views of QoS or the fact that different components are managed by different organisations. However QoS management of components in a Web delivered service does not indicate how a service actually performs to the end-user. This leaves the Service Provider not knowing how the service performance and availability are perceived by the end-user. What is suggested is that monitoring the service as perceived by the end-user offers a true reflection of service availability and performance and provides a valuable diagnostic tool for the Service Provider as to how the service performs overall. Finally the objectives of this dissertation, the technical approach taken and the dissertation structure were outlined.

2 TECHNOLOGIES FOR END-USER QOS

2.1 Introduction

In this chapter, a number of key issues and enabling technologies that are relevant to this dissertation are examined. Firstly, the area of QoS and current approached to Service Management are discussed. Following on from this, Service Level Agreements are discussed including a briefing on the components of a SLA. Concluding the chapter is a review of the technologies for monitoring QoS and describing SLAs.

2.2 Quality of Service

In the following section, Quality of Service (QoS) is discussed as it is a mechanism that allows some predictability to be introduced to services delivered via the Web at a number of different architectural levels such as IP and application level QoS.

2.2.1 What is Quality of Service

As stated in the introduction, Quality of Service has a number of different meanings depending on the context we are talking about. Essentially Quality of Service means providing consistent, predictable data delivery service. In other words, satisfying customer application requirements. Once can refer to QoS at a number of different levels including network, application and end user QoS. For example, implementing QoS so that specific users can receive a better that “best effort” service from a Webserver has been investigated [4].

2.2.2 Why the need for QoS

One may ask the question as to why QoS is needed at all. For example, in networks there is an increasing investment in new technologies that are increasing bandwidth available for use

and in the area of web hosting, technologies such as load balancing provides the means to handle huge number of simultaneous requests to services. Although ever-increasing bandwidth and over-provisioning can help to improve conditions, there are a number of reasons why the introduction of QoS is still necessary.

- Networks, systems and applications are susceptible to congestion and overload that can affect data throughput because application traffic is unpredictable by nature. Since these applications often share the same resources at the same time, congestion is often the result. Web sites can be hit due to a massive increase in traffic or what is known in the Internet world as "the Slastdot Effect" [8]. This often results in what seemed like over-provisioned servers and networks to quickly appear insufficient due to sheer volume of attempts to reach a site simultaneously. There are simply too many users now on the web to deliver a first class service all the time based on best effort.

- Different applications have different requirements for throughput, reliability, delay and jitter. Some service are more elastic that others. Although users are normally prepared to put up with delay with elastic applications[1] because it is expected to be delivered later in the day and picked up some other time, one may send an urgent email which can be treated as a real-time or inelastic application [9]. So it would be more efficient if one could segment traffic based on its requirements for delay, jitter etc. Even on relatively unloaded IP networks, delivery delays can vary enough to adversely applications that have real-time constraints [3].

- Certain users are willing to pay for a guaranteed service, while others are willing to suffice with a "best effort". Since QoS provides value in the service, it implies the need for **accounting and billing.**

The last of the three points above is of vital important. QoS is how network and application Service Providers as starting to distinguish themselves from each other. There is even the emergence of QoS enabled Web Servers. Empirical evidence suggests that overloaded servers can have significant impact on user perceived response times. Furthermore, FIFO scheduling done by servers can eliminate any QoS improvements made by network differentiated services. Consequently, Server QoS is a key component in delivering end to end predictable, stable, and tiered services to end users [4]. This concept of QoS for Web Servers will be investigated later in more depth.

QoS by definition means that some users are getting a better service than others. Therefore QoS requires a policy when there is contention so that the network or application knows which users are entitled to which services or in the context of Web delivers services, which connections to a web server are for premium services. It is not possible to enforce a policy if one cannot establish the identities of network or application users, so another function that is required is authentication. Other functions of QoS include service monitoring and configuration.

2.2.3 IP QoS

Generally, IP QoS can be broken down into two different types. These are Reservations and Prioritisation QoS [2].

2.2.3.1 Reservation Based

Integrated Services Architecture (IntServ) - The Integrated Services Architecture being defined by the IETF is intended to transition the Internet into a robust integrated-service communications infrastructure that can support the transport of audio, video, real-time, and classical data traffic. Network resources are apportioned according to an application's QoS

---

[1] An elastic application is an application whose QoS requirements are not highly constrained and for which delays are acceptable in its usage. An example of a highly elastic application would be email whereas video conferencing would be considered inelastic.

request, and subject to bandwidth management policy. RSVP provides the mechanisms to do this, as a part of the IntServ architecture.

RSVP supports two different types of reservation:

- **Guaranteed**: This comes as close as possible to emulating a dedicated virtual circuit. It provides firm (mathematically provable) bounds on end-to-end queuing delays by combining the parameters from the various network elements in a path, in addition to ensuring bandwidth availability.
- **Controlled Load**: This is equivalent to "best effort service under unloaded conditions." Hence, it is "better than best-effort," but cannot provide the strictly bounded service that Guaranteed service promises.

2.2.3.2 Prioritization Based

Differentiated Services Framework (DiffServ): The Differentiated Services Framework being defined by the IETF is intended to meet the need for relatively simple and coarse methods of providing differentiated classes of service for Internet traffic, to support various types of applications, and specific business requirements. The differentiated service approach to providing quality of service in networks employs a small, well-defined set of building blocks from which a variety of services may be built. Network traffic is classified and apportioned network resources according to bandwidth management policy criteria. To enable QoS, classifications give preferential treatment to applications identified as having more demanding requirements.

DiffServ currently has two standard per-hop behaviors (PHBs) defined that effectively represent two service levels (traffic classes):

- ***Expedited Forwarding*** (EF) EF minimises delay and jitter and provides the highest level of aggregate quality of service. Any traffic that exceeds the traffic profile (which is defined by local policy) is discarded.
- ***Assured Forwarding*** (AF): Has four classes and three drop-precedences within each class. Excess AF traffic is not delivered with as high probability as the traffic "within profile," which means it may be demoted but not necessarily dropped.

2.2.3.3 What are the primary IP QoS Parameters?

Generally, QoS parameters can be broken down into the following [3].

- **Latency** - The time between a node sending a message and receipt of the message by another node.
- **Jitter** - An aberration that occurs when video or voice is transmitted over a network, and packets do not arrive at its destination in consecutive order or on a timely basis, i.e. they vary in latency.
- **Bandwidth** - A measure of data transmission capacity, usually expressed in kilobits per second (Kbps) or megabits per second (Mbps). Bandwidth indicates the theoretical maximum capacity of a connection, but as the theoretical bandwidth is approached, negative factors such as transmission delay can cause deterioration in quality.
- **Packet Loss** - Example: 1% or less on network-wide monthly average packet loss.
- **Availability** - Example: 99.9% premises to service provider.

2.2.4 Application Level QoS

Generally QoS is applied to applications that have specific constraints for bandwidth, jitter and delay. The techniques mentioned previously are able to address these issues. However application themselves can also benefit from QoS. "*As Internet usage grows it has become apparent that non-isochronous applications such as delivering static and dynamically generated web pages can also benefit from QoS*" [4]. Clearly, there is little point in establishing network level QoS if the end application that is being accessed has no priority on which request to processes first. Some of the results that were discovered in this research are as follows:

- servers were currently a significant component in end to end delay
- there are several trends that are increasing server latency time for sophisticated Internet applications including
    - Flash crowds can overload a popular site leading to poor response times or even denial of service

- Current process queuing techniques on servers mean that processes take so long that clients simply disconnect
- New technologies such as SSL, Java, Database transactions and middleware
- Media is also becoming rich with larger and more images and even voice and video being requested

It also highlighted the response rate versus the HTTP GET rate plotted for a typical web server. As expected, when measured from an HTTP request perspective, the response rate grows linearly until the server nears maximum capacity as follows.

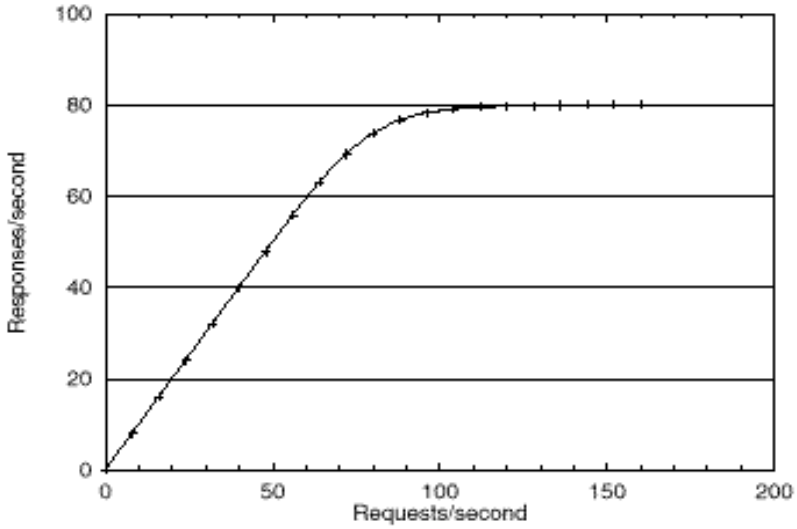


Figure 1: Graph illustrating Requests/sec Vs Responses/sec [4]

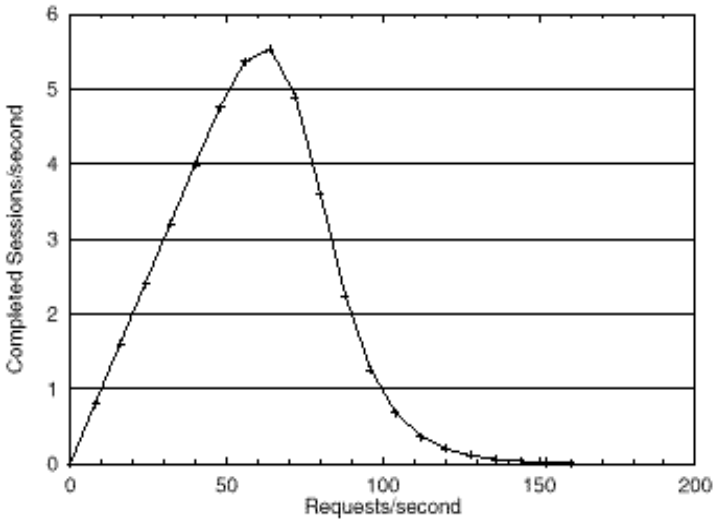


Figure 2: Graph illustrating Requests/sec Vs Completed Sessions/sec [4]

What was even more interesting was when the same data is analysed from a client's perspective. For a series of HTTP requests that would be typical of that involved in an online transaction or a session as they are referred to, the session throughput collapses rapidly as the server becomes busy due to queuing and congestion on the server. Since generally on E-commerce sites, longer sessions are more likely to be due to a purchase going on, this represents a significant issue. One may say that why could these issues not be addressed through over-provisioning? If one looks at the growth rate in the number of clients for web-based applications, it quickly becomes apparent that the demand curve is such that static provisioning will not support the demands that could be placed on them.

To deal with this issue, QoS was implemented by developing a modified version of the apache server. It allowed each request to be classified according to two methods

1. User Class Based Access: This uses the clients IP address, cookies or some other method to distinguish the requests and give appropriate priority to it

2. Target Based: Supports URL based prioritisation so that for example when a user starts to purchase a produce it receives higher priority

The previous section has presented a brief synopsis of how QoS is implemented on a number of levels. It is worth mentioning that architectures have been developed that attempt to enable end-to-end QoS for applications that have highly constrained requirements, in particular multimedia applications[10] [11]. Although these architectures provide a means to provision QoS from the network up to the application on an end-to-end basis, they are not specifically targeted at the most common application delivery framework for the foreseeable future, which is Web based delivery of applications.

Finally, there is one final type of QoS that has not been discussed at all. This is Quality of Service as perceived by the end user and is of particular relevance in the context of web delivered applications. End user QoS encompasses the failure or success of all other forms of QoS from server to client in the service delivery chain.

2.3 Current Approaches to Service Management

A number of current approaches to service management will be discussed next. They offer some insight into current methods of managing services and determining service availability. These examples are not specifically Web based services but highlight the concepts and issues involved.

One approach to service management is to manage services as a single unit, not just as individual components. For instants, an Internet Service Provider (ISP) might provide an e-mail service on one of its servers to its customers. For this service to operate within specified parameters, all of the components within that host on must be operating.
Research suggests that services should be modeled as a tree structure [5]. For example an E-mail service would have the following structure.

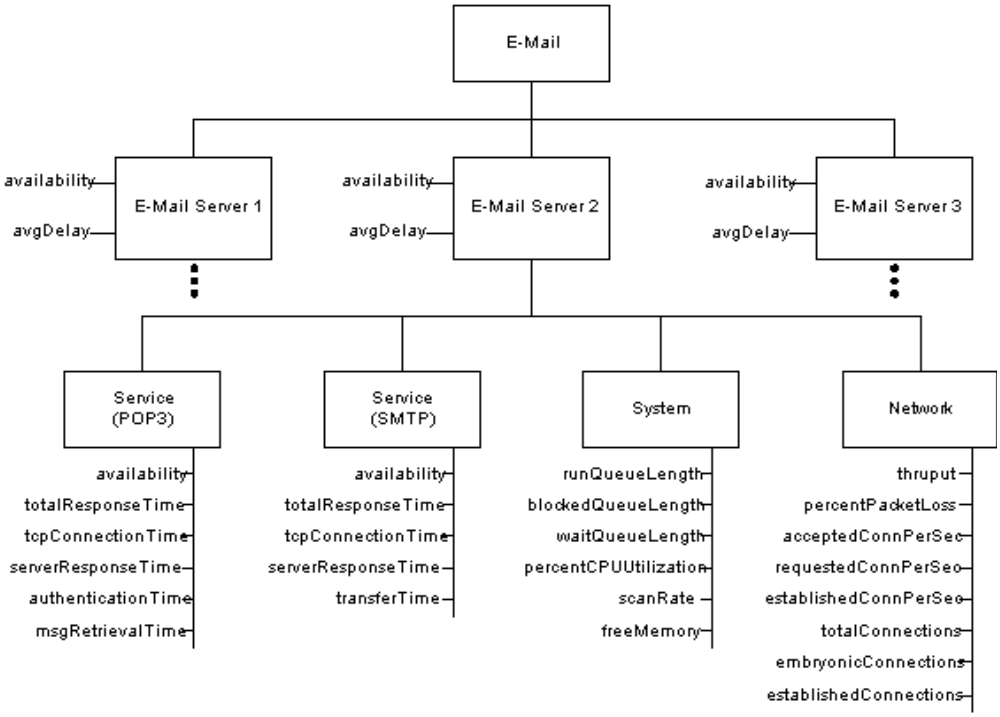


Figure 3: Model of an Email system [5]

However, this model fails to take into account the interdependencies a service may have on other hosts and services. If one was to consider an E-mail service, it has many components on which it depends including DNS, IP connectivity and SMTP servers. Failure of any of these components often means that the whole service fails to operate.

Two separate, but similar models that address this focus on actually determining the total availability of the service defined taking into account all the components the service depends on [6, 7]. The latter is more focused on actually determining the total availability of the service defined taking into account all the components the service depends on. The first targets the issues of how to compose a service model automatically. It approaches the problem by defining templates for typical ISP services and then combining this with some auto-discovery tools which are then passed through an engine that can generate the service model instance.

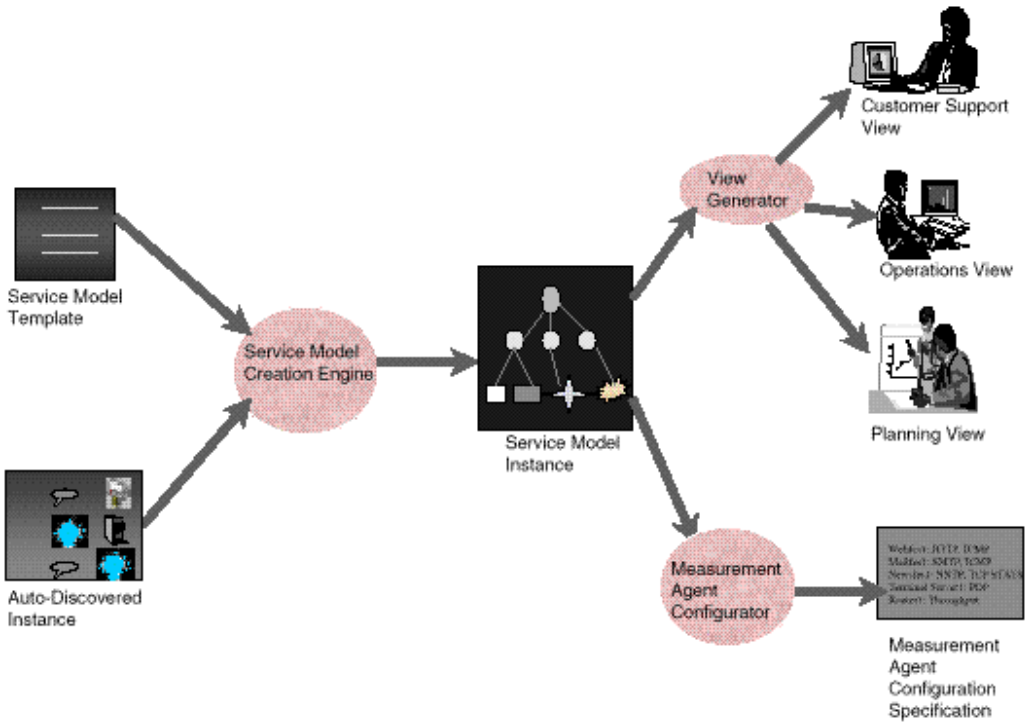


Figure 4: Illustration of Service Instance creation engine [6]

Again, what is clear is the idea that services are composed of components and some method is required that can provide an overall view of the service performance. The focus is on how the service performs overall.

2.4 Service Level Agreements

2.4.1 Overview of Service Level Agreements

Bearing in mind what has been discussed in the previous sections, the first point to be made clear is that customers do not care how a service is composed -- to them the quality of the end service is what is important [12]. Generally a Service Level Agreement can be defined as a contract between a Service Provider and a Customer that guarantees specific levels of performance and reliability at a certain cost [12]. Typically the Service Provider is the party providing a service and the Customer is the party paying for and receiving the service and for any one SLA there will be a customer and a service provider.

SLAs are important in the fact that they provide a means in the context of service management to formally define the behavior and quality of the service being delivered. It can also include details on performance monitoring and reporting of the service being delivered, as well as actions to be taken when performance or availability do not meet the levels specified in the SLA.

SLAs until recently, often concentrated on addressing availability and customer support guarantees, such as guarantees on helpdesk problem resolution time on service outages. With the proliferation of service outsourcing, where an organisation outsources all or portion of a service to third party vendors, the organisation will usually require guarantees on the performance and availability of the service that has been outsourced.

One of the reasons performance has become a critical factor in Web delivered services is the fact that the Web is based "best effort" mechanisms of the Internet and the components in the delivery chain so QoS has become of critical importance. Generally the issue with web-based service is application burnout where the service simple slows down to the extent that users can't utilise the service anymore [13].

2.4.2 SLA Negotiation

"SLA negotiation takes place between customers and Service Providers during service ordering. It enables the Service Provider and customer to formally and legally state the responsibilities of each with regard to the service(s) being ordered by the customer from the Service Provider" [14].

There can be a variety of types of SLA negotiation. In the case of on-line negotiation, the interaction to formalise the specification of the SLA could be done via a web front end that generates the appropriate SLA. Off-line negotiation would involve person-to-person communication, such as telephone, meetings, etc. On-line SLA negotiation is more applicable for pre-defined services where the customer can fill in a template on-line. Off-line negotiation is more likely to be required for more complex services [14].

SLA negotiation is concerned with three main areas

- The actual service(s) details of the service(s) being purchased
- Problem handling: what should happen when a problem occurs
- Proof of compliance: how service performance is monitored and reported.

2.4.3 Items contained in an SLA

The contents of a Service Level Agreement can be broken into a number of different sections as follows [14]:

- **General Items:** These typically include a unique identifier (within the Service Provider's domain) for a particular SLA, definition of the terms used in the SLA and identification of the parties to the SLA. Other items will include detailed textual definition of the service(s) covered by the SLA and, procedures to be invoked on violation of SLA guarantees

- **QoS Aspects:** Clear and unambiguous definition of the service-independent parameters for each service together with performance metrics for each individual parameter required for SLA compliance.

- **Trouble Handling Aspects:** Procedures to be used to report a problem to the Service Provider or a third party

- **Monitoring and Reporting Aspects:** Includes specification of reporting on SLA compliance, service performance reporting period and reporting frequency.

Other items contained in a SLA may include accounting and discounting agreements and security aspects. However, in this dissertation, QoS monitoring and reporting aspects will be the items mostly focused on.

2.5 HTTP

2.5.1 Introduction to HTTP

The Hypertext Transfer Protocol (HTTP) defines how client and server applications communicate in order to transfer hypertext documents and other resources located on the Internet although it is not specifically limited to the TCP/IP protocol stack. The protocol is independent of the type of resources transferred so data may be text, sound, images, query results from a database or even an application to be executed on the client machine.

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred [15].

The format of a typical HTTP request-response is as follows

1. A client opens a connection with a server.
2. The client sends a request to the server. This request consists of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible entity body content over a connection with a server.
3. The server returns to the client a status line, header information, and possibly some entity metainformation and an entity body.
4. The server closes the connection.

HTTP is a stateless protocol in that it does not maintain any connection information between transactions although HTTP/1.1 does support persistent connections so a connection does not have to open each time to the fetch each URL and in doing so reduces the load on the server. This means that if a hypertext page has embedded references to other resources, the

client can send multiple requests over the same connection for those resources. It also allows a client to make requests without having to wait for each response before it makes the next request, allowing a single TCP connection to be used much more efficiently. Technologies for SLAs and QoS Monitoring

2.5.2 XML

One of the primary focuses of this dissertation is to investigate a method of describing how an HTTP delivered service should perform. XML, the Extensible Markup Language, is emerging as a key technology in describing the structure of data. However, XML is not a language as its name suggests, it's actually a metalanguage or a language used to describe other languages [16]. It allows the creation of customised (Extensible) markups so unlike HyperText Markup Language (HTML) where the markup is limited and can only describe one type of document, XML allows the creation of customised markups. One key feature of XML in comparison to HTML is XML tells what data signifies rather than how to display it. XML is based on the SGML (Standard Generalized Markup Language), which is the international standard for defining descriptions of the structure and content of different types of electronic document. However XML has been designed so that is simpler to use that SGML with the aim of promoting the use of SGML on the Web.

2.5.2.1 DTDs

Document Type Definition (DTDs) specifies the tags that can be included in an XML document, and the valid arrangements of those tags. The DTD specification is actually part of the XML specification, rather than a separate entity. However a DTD it optional and XML documents can be created without one. DTDs help avoid creating invalid XML structures and allow the verification of the XML structure of a particular document.

2.5.2.2 XSL

As stated already, XML specifies how to identify data, not how to display it. HTML, on the other hand, tells us how something should be displayed without identifying what's displayed. Extensible Stylesheet Language (XSL) specifies what to convert an XML tag into

so that it can be displayed in another format such in HTML [17]. Different XSL formats can then be used to display the same data in different ways, for different uses. This provides the ability to render the same information differently for different situations of for different devices.

2.5.2.3 Using XML to describe a SLA

Increasingly XML is emerging, as the de-facto standard for describing information content and the area of SLAs is no exception. What are the advantages of using XML to describe a SLA? First of all, once a DTD is defined, it allows the creation of SLAs in XML and both computers and humans can easily understand XML due to its structured form. It presents the possibility for linking and merging multiple components of SLAs to define a higher level SLA. XML is particularly suitable as a transport encoding format and the fact that SLAs often need to be transported between different systems provides another advantage in using it. It also supports different views of the same data through the use of XSL [16]. This is useful in the fact that some items of a SLA might not be that interesting to a customer but would be interesting to the Service Provider.

2.5.3 HTTP Proxies and Caching

HTTP proxies are an integral part of the Web today. A proxy is defined as an intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy must implement both the client and server requirements. A "transparent proxy" is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. A "non-transparent proxy" is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering. Except where either transparent or non-transparent behavior is explicitly stated, the HTTP proxy requirements apply to both types of proxies [15].

Although proxies provide various capabilities that improve the end-user experience with features such as caching, they are examined in this dissertation for the ability to analyse HTTP request and response information for requests that are directed through them.

2.5.3.1 Performance and Availability Analysis Capabilities of Proxies

When user-agents are configured to pass requests through a proxy, the proxy emulates the target server's interaction with the user-agent. It also emulates the user-agents interaction with the target server. Therefore the proxy is both a client and server. The fact that a proxy provides a means to capture these requests and replies means that it offers the possibility of analysing various types of information pertaining to a HTTP request. Using proxies to capture information in this way has been used for a number of years. The free caching proxy software, *Squid*, offers a vide variety of scripts to analyse and summaries requests to it. Other research tested throughput to a remote HTTP proxy [18]. In this case, the proxy measures the time that a proxy thread receives a request, to the time the service of the request is completed. It also measured the amount of data logged by each request to throughput could be calculated. Such research focuses on the performance aspects of HTTP, and does not attempt at measuring the availability of an HTTP delivered service. However it does offer some interesting insights into the issues involved.

2.5.3.2 Proxy the End-User Experience

As was stated in the introduction of this dissertation, the possibility of being able to capture the performance and availability of a service as perceived by the end user offers a really valuable way to analyse a Web delivered service. What the author suggests is that a client side proxy could be used to capture information that can then be used to determine how a service performs from the end-user perspective. Where to position such a proxy that can intercept service requests is work examining further. The proxy can in theory be placed anywhere between the HTTP server that delivers a service and the end-user's browser. However, the nearer the proxy is to the end-user's browser the more accurate the timing information will be. So the advantages of placing the proxy on the client side are:

- The latency between the browser and the proxy are negligible. This means that the proxy receives the request from the browser with very little latency and browser receives the

reply almost immediately once the proxy receives it from the origin server. This results in any timing information that is logged accurately reflecting the instances that requests from and replies to the browser are performing at.

- Any connection problems to the service can be detected and logged. If the proxy is located remotely and the host that the browser is running on is disconnected from the proxy in some way, any requests to the service, or failures, will not be logged.

Clearly HTTP proxies and caching proxies in particular, offer significant benefits including a saving in bandwidth and reducing latency which result in an improved user experience. A HTTP proxies provide the ability to intercept any requests and replies the end-user makes to a Web delivered service and could be used to indicate the performance as experienced by the end-user, particularly when placed on the same host that the browser runs on.

2.6 Summary

This chapter has covered a number of key issues and areas including the Quality of Service, current approaches to Service Management. Service Level Agreements were investigated including a briefing on the items that are typically contained in a SLA. Finally an overview of technologies that allow monitoring QoS and the description of SLA were covered.

3 REQUIREMENTS

3.1 Introduction

This Chapter first revisited the motivations of this dissertation and the primary objectives defined in Chapter 1. It describes the use of HTTP as a service delivery mechanism. The potential sources of performance and availability issues that HTTP is prone to are highlighted. Next the monitoring requirements of HTTP delivered services are analysed including what parameters should be monitored. Finally the issue of how to monitor services so as to be able to capture the availability and performance as perceived by the end-user is addressed.

3.2 Motivations Revisited

Before going further, it is worth revisiting the motivations of this research.

3.2.1 Motivations

The primary motivations are:

- The emergence of Application Service Providers and the use of the Web as a delivery mechanism for services. One of the key issues is can services that are delivered over the Web perform comparably with traditionally hosted applications
- Quality of Service mechanisms for network, application and other components used to deliver services can help to guarantee the overall performance of a service. However, there is little point in establishing network level QoS if the end application that is being accesses has no priority on which request to processes first. Therefore appropriate QoS needs to be implemented on all components. Another issue is that the Service Provider does not always manage the

components that a service is delivered over and so cannot tell if SLAs for QoS in components in the delivery chain are being met.

- QoS mechanisms offer little indication as to how a service is performing from the end-user perspective.
- End-user performance is the ultimate indication of overall service performance but the Service Provider has no way of getting this information to diagnose problems in service delivery.
- Traditional SLAs that define overall service performance, do not contain meaningful parameters to specify how a service should perform from the end-user perspective.

3.2.2 Objectives

The primary objective of this dissertation is to develop a framework that can support QoS monitoring and Service Level Agreements for Web delivered Services. This has three sections to it.

- To design a client side proxy that supports automatic configuration, monitoring and logging of availability and performance parameters as perceived by the end user of Web delivered services
- The design of a SLA template specifically for Web delivered services that will map these parameters captured by the proxy in a meaningful way so that it is possible to specify how a service delivered over HTTP should perform, from the end-user perspective
- To implement SLA verification and feedback to the Service Provider and Customer so as to notify them of SLA compliance.

Based on the objectives above the requirements can be broken into 3 main sections:

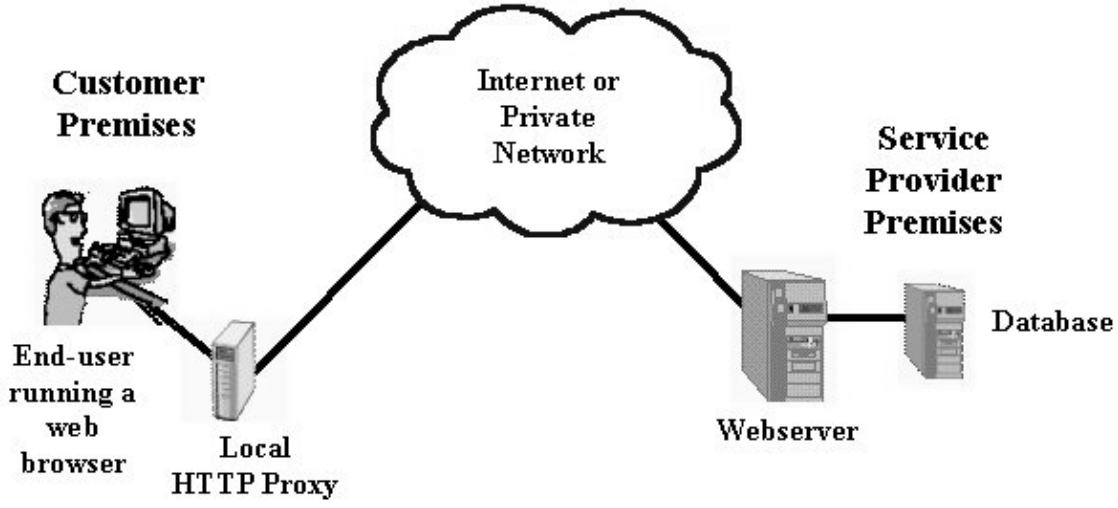
1. Client-side component to support logging of appropriate availability and performance parameters
2. Design a suitable SLA for Web delivered services
3. SLA verification and feedback to the Service Provider and Customer

3.3 Service Delivery using HTTP

Service delivery via HTTP will now be examined including what the monitoring requirements of such services might be and the methods available.

3.3.1 Anatomy of A Web Delivered Service

Services and the delivery mechanisms for them are changing from locally managed and specific application interfaces for each service, towards remotely managed and Web delivered accessible via a web-browser. A service can typically be any information-based service that is provided to a customer. For example, the Web-based Email services that are provided free by Yahoo and Hotmail are typical examples of web delivered services. The following illustration is a simplified overview to how a typical service is provided via HTTP.



z

Figure 5: Illustration of path taken for a HTTP request and response

Taking a high level look and ignoring details of routing and Domain Name System (DNS) requests, let's examine how a typical transaction occurs when a end-user accesses the service.

1. The user enters a URL in a web-browser to access the service. The URL may intentionally or unintentionally give some indication as to the purpose of the resource identified in the URL. For example, if the service is a web-based email service, a URL as follows may return a HTML page listing mail in the users inbox:

   http://webmail.cs.tcd.ie/showInbox

   However, in most cases of services delivered via the web, the user go to a higher level URL that's easy to remember such as: *http:// webmail.cs.tcd.ie*
   This URL may provide a general listing of the service functions formatted in HTML and represented as hyperlinks so that the list of functions is easy to navigate for the user. In the case of a web-base email service these functions and the URLs they map to could be as follows:

   - Login to mail => *http://webmail.cs.tcd.ie /loginToMail*
   - Logout of mail => *http:// webmail.cs.tcd.ie/logoutOfMail*
   - Delete mail => *http:// webmail.cs.tcd.ie/deleteMail*
   - Send mail => *http:// webmail.cs.tcd.ie/sendMail*
   - Compose mail => *http:// webmail.cs.tcd.ie/composeMail*

These are very simplified URLs in the sense that different service functions are represented by different resources. However it could be that different service function are accessed by the same resource and identified by different queries "?" in the URL.

For example the URLs:

http://webmail.cs.tcd.ie/webMail?readMail
http://webmail.cs.tcd.ie/webMail?sendMail

Both of these URLs identify the same resource, */webMail*, but the query portion of the URL identifies to the server the function that the user wants to access.

It is also possible to pass parameters using the POST method from a HTML form that can then be used by the server to determine the service function the user is wants to access. In this case the URL will not indicate that these parameters have been sent to the server as POST sends form parameters in the body of the submission [19].

2. If a local HTTP proxy is used in the customer premises, the request is passed to the proxy to be handled. If this local HTTP proxy has a parent proxy, then it may in turn pass the request on to the parent request to be dealt with.
3. Once the request reaches the highest level proxy, the request is then passed to the Service Provider. This may take place over the Internet or perhaps via a leased line directly to the Service Providers Premises.
4. When the request reaches the Service Provider's web-server, it may be a request for a static document in which case the requested resource can be returned immediately. In the case of dynamic data such as retrieving the listing of mails in a user's inbox, the web-server may query information from a database that is to be returned to the user via the web-server.
5. The reply to the request is usually returned to the end-user via the path and proxies that the request was sent on. The resource may be cached, if appropriate, in any of the proxies on the return path to the client. This may allow the resource to be accessed from the proxy later by other users in the customer premises without having to retrieve the resource again. The reply is the formatted by the user browser to display the information requested.

3.3.2 Where can Availability and Performance Issues Occur

Even looking at the relatively basic example above, failures or performance impacts may result from any of the following:

- Any network component such as the connection between the end user and the local proxy, the web-server and database or the local proxy and web-server
- Any proxies used in the service delivery chain may fail or impact performance
- The web-server(s) that the service is provided via may fail or impact performance

- Any backend components such as databases that the service depends may fail or impact performance

How can performance problems be overcome? QoS technologies such as WebQoS and Diffserv can help solve application and network related performance issues [2, 4]. However for QoS guarantees to be possible, QoS needs to be implemented in all components in the service delivery chain. Clearly, there is little point in establishing network level QoS if the end application that is being accesses has no priority on which request to processes first. The problem for the Service Provider is that it does not manage all of the components in the service delivery chain so establishing appropriate QoS for all the components can be difficult between different organisations.

In the context of the availability issues addressed above, redundancy may assist in making a service more reliable in that if one component fails in the delivery chain, another can take over to allow the service to continue. However, redundancy is usually not applied to all components in a service deliver delivery chain, if for any reason, the cost would be prohibitive. So if the customer's network connection to the Service Provider fails, the customer will not be able to access the service. However, the Service Provider may not be aware that the customers is experiencing difficulty accessing the service.

Of course, guaranteed levels of QoS and availability of all the components gives no indication as to how the service performs from the end-user perspective. Tackling this problem will be discussed next.

3.4 Monitoring HTTP Delivered Services

Having looked at how HTTP services are delivered, the next area to investigate is how can HTTP delivered services be monitored to give an accurate reflection of the service as perceived by the end user. Looking at types of Parameters to Monitor for Services Delivered by HTTP, they can be broken into two types:

- Performance parameters

- Availability parameters

3.4.1 Service Performance Parameters

For performance parameters, they can typically be broken down into parameters that are concerned with timing information such as the time a request is made and data-transfer information such as the length of a reply message body. Combining both of these types of parameters offers the possibility to derive other parameters such as throughput and latency.

3.4.1.1 Base Request/Reply Performance Parameters

The following are suggested base parameters for HTTP performance.

Parameter Name	**General Description**
RequestTime	The time in milliseconds that the request is issued
RequestBodyStartTime	The time of starting to send a message body (if any) associated with a request
RequestBodyEndTime	The time in milliseconds of finishing the sending a message body (if any) associated with a request
RequestContentLength	The content length of the request message body (if any)
ReplyTime	The time in milliseconds that the reply headers are received
ReplyContentLength	The content length of the reply message body (if any)
ReplyContentType	The content type of the reply message body (if any)
ReplyBodyStartTime	The time in milliseconds of starting to receive a message body (if any) associated with a reply
ReplyBodyEndTime	The time in milliseconds of finishing the receipt a message body (if any) associated with a reply

Table 1: Base Performance Parameters

Although, there are other parameters that could also be monitored such as the IP address of the client making the request, the ones that are identified above are key parameters for analysing the performance of HTTP.

3.4.1.2 Deriving Performance Parameters

Having defined base performance parameters, lets look at how these parameters can be used to determine the performance of an HTTP transaction. The base performance parameters that are listed above can be used to derive more useful performance parameters as follows:

Derived Parameter	Description	Derived By
EntityUploadRate	The rate of upload to the server of the request message body	requestContentLength/ requestBodyEndTime- requestBodyStartTime
HeaderResponseTime	The time elapsed from sending a request and the request message body to a server, any processing by the server and the receipt of the reply header by the client.	replyTime- requestTime
EntityDownloadRate	The rate of download from the server of the request message body	replyContentLength/ replyBodyEndTime- replyBodyStartTime

Table 2: Table of Sample Derived Performance Parameters

Of course, there are numerous other parameters that could be derived. However, the derived parameters listed above give an example of what's possible.

3.4.1.3 Interpretation of Derived Performance Parameters

Further analysis of derived parameters above will now be undertaken to indicate their usefulness and situations there they are applicable.

entityUploadRate: This gives a good indication as to the connectivity between the client and server and the ability of the server to process information sent to it. Typically this could only be used when the request has a message body such as when a POST or PUT method is used with an HTTP request.

headerResponseTime: The time between sending a request to the server, any processing of the request by the server and the receipt of the reply headers. However, if the request has an entity body this will impact on the time to send the request to the server.

entityDownloadRate: Indicated how fast the server and the network connection between the server and client can return a reply message body to the client. Again some replies may not have an entity body as would for with a status-code of 1xx (informational), 204 (no content), and 304 (not modified) responses.

So as has been established, not all derived parameters can be used in all situations. However, the option to derive a suitable parameter for a particular situation still exists enabling useful information to be generated.

3.4.2 Service Availability Parameters

Availability parameters are parameters that are use to indicate the availability of a service delivered to the end-user. These parameters can be broken down into three main types of parameters that can indicate service failure.

- Connection failures to the service such as a network failure or failure to resolve a server name. These result in the client not being able to open a HTTP connection to the server
- Server Errors occur when the client can make a connection a HTTP server but a error in the server occurs and results in a reply status-code of the 5xx variety
- Service Errors where a backend component that the service depends on (such as a database), fails and results in the service failure. No mechanism is implemented in HTTP to indicate this as it is not really related to HTTP protocol itself, but more to the service that is being delivered over HTTP.

3.4.2.1 Availability Parameters

The following are suggested parameters for HTTP availability that can identify if a service has failed:

Parameter Name	General Description
RequestException	Any errors that occur in making a request such as failure to connect to a server or DNS lookup failures
ReplyStatus	Use to capture the status-code. 5xx reply status indicate a server error
ServiceErrorHeader	Use to capture service error headers that indicate failure in a backend component that service delivery depends on. These headers are required as backend component failures will return valid HTTP responses that cannot be detected as an error. These headers must be agreed with the Service Provider so that they can be detected in HTTP responses from the service

Table 3: Availability Parameters

Although, there are other parameters that could also be monitored such as the IP address of the client making the request, the ones that are identified above are key parameters for analysing the performance availability of HTTP.

3.5 Parameter Capture

Having defined base performance and availability parameters, the next issue is where to capture these parameters to give an accurate reflection of the end-user experience. The options available include

- Monitor on the server-side
- Monitor between the browser and server by using a proxy: The implications of where to place such a proxy have been discussed already.

3.5.1 Server-Side Monitoring

Server-Side monitoring in the context of Web delivered service involves capturing information on HTTP requests to and replies from the server, at the server itself. The problem with server-side monitoring is that is does not provide an accurate representation of the end-user experience because failed attempts to the server cannot be detected.
Another issue with server-side monitoring is the performance implication of monitoring HTTP to this level of detail for each request and since some customers may not require such detailed monitoring it could be an unnecessary overhead.

3.5.2 Monitoring via a Proxy on the Client-Side

The other alternative to server-side monitoring is monitoring between the server and the browser by forcing HTTP requests to the server through a proxy. Ideally the location of the proxy should be on the client, the reasons for locating the proxy on the client have been discussed already. The implications of monitoring by a proxy include:

- Not all services that are accessed via the proxy may require monitoring. The proxy must have the ability to configure, reconfigure and disable monitoring of a service.
- It results in another layer of indirection and there may be performance implications in monitoring performance
- Service performance information needs to be passed back to the service provider and perhaps a third party. The proxy needs to know what information to send and at when to send it.
- Ideally the proxy should have the ability to detect service errors and backend component failures that impact service availability but return a valid HTTP reply.

So despite the advantages of using HTTP proxies for service monitoring, there are significant obstacles to overcome if they are deployed.

3.5.3 What information needs to be captured?

What information does each party involved in a Web delivered service what to know about how the service is performing for the end-user?

For the customer, they may wish to know the following:

- What percentage of requests to the service are performing as specified in the SLA: **service performance information**
- What percentage of requests to the service are failing: **service availability information**
- How many requests have been made to the service in the last hour: **service usage information**

For the Service Provider, information that they might want to know includes:

- What percentage of each customer's requests comply with the SLA for that service: **service performance information**
- What are the most common types of errors in service delivery in the last month: **service availability information**
- How many requests have a particular customer made to the service in the last day: **service usage information**

So to be able to verify **performance** and **availability** implies having **usage** information available also.

3.5.4 QoS Mappings

One of the difficulties already highlighted with how SLAs and QoS management are traditionally approached is the lack of meaning to the end-user on how the service is perceived to perform. This has been partly due to the focus on managing services in the context of overall availability and performance of the components that the service depends on. The other reason has been the difficulty in monitoring and reporting service performance and availability as experienced by the end-user. Even with the means to capture performance and availability parameters for a service delivered over HTTP, some interesting problems are

presented. First of all, what performance and availability parameters are to be calculated and what are they to be calculated against? The end-user of a service delivered via the Web views performance in terms of how fast the response to a request arrives and typically this means how soon the complete object appears in their browser. Of course, this is over simplifying the complexities involved in a HTTP transaction. Service functionality is usually delivered by unique URLs for Web delivered service and these URL will have different request/response rates depending on the amount of data passed to and from the server and the processing required on the server-side. For example, the request may have an entity body associated with it as occurs with the submission of the contents of an HTML form to the server using a POST method. This data associated with the request entity may result in an increased delay in the complete request being received at the server. There is typically a delay associated with processing the request on the server and this may vary depending on what resource has being requested. The requested resource may involve some queries to a remote database for example and this may also significantly increase the processing time of the request on the server before the reply is sent to the client. Finally, the reply will usually contain content of some type and depending on the size, will again have an impact on when the complete reply is received at the client.

This implies that:

- The amount of data sent with each request varies which may impact when the complete request is received at the server
- The processing time of a request at the server varies depending on the resource requested. The content of reply messages may be dynamically generated while others may be returning data representing static objects such as images on the server file system
- The amount of data sent with a reply can vary for a request, even for the same resource, in the case where it is dynamically generated.

So the specification of performance parameters for services needs to flexible to take into account the circumstances of how the end-user interfaces with a service(s) via HTTP. As was mentioned already, some derived parameters are more suitable to specific situations. For

example, stating that a GET request to a resource should returns the complete reply body in one second from the time of issuing the request may not a reasonable if the amount of data in the reply body can vary from 1 Kb to 1 Mb. A more appropriate parameter in this case may be that the entity download rate is greater that 10 Kbps.

3.6 Conclusion

This Chapter first revisited the motivations of this dissertation and the primary objectives defined in Chapter 1. The use of HTTP as a service delivery mechanism was investigated, as were the potential sources of performance and availability issues. Leading from this, monitoring of HTTP delivered services and the parameters that such service monitoring should capture were defined. Finally the issue of how to monitor services so as to be able to capture the availability and performance as perceived by the end-user was addressed.

4 DESIGN

4.1 Introduction

The previous chapter highlighted the requirements for monitoring Web delivered services and identified parameters to capture and the options that exist to capture them. This chapter will first examine the possible architectures to support such service monitoring and then discuss the design of the components that enable such monitoring to take place.

4.2 Architectures for Service Monitoring

Having determined the type of information that should be captured for a service, the use of that information can be varied. In most cases, information captured on a service will be used by:

- The Customer to determine if a Service is performing as specified in a SLA
- The ASP provider to determine if the service it's providing is meeting the SLA with it's customer and possible to be alerted to service faults

A number of architectures to monitor Web delivered services will now be examined. The strengths and weaknesses will be highlighted for each one. Some architectures result in a more efficient and flexible Service monitoring and reporting. For example, if a proxy is used to monitor a service and has access to a SLA for that service, a certain amount of data analysis can be done by the proxy and then appropriate action taken depending on the result of that analysis. If the proxy does not have the SLA it may have to return the information logged for all HTTP requests to the party monitoring the service performance before performance and availability issues can be highlighted to the ASP and customer. This also results in a greater amount of data being transferred and analysed remotely, which may not be a very scalable solution. This just highlights some of the choices that must be made when deciding on how information is exchanged between a Customer, Service provider and a third party

4.2.1 A Single ASP and Multiple Services

The first architecture that is presented is where the customer installs a proxy provided by the Application Service Provider on the end-users host. The web-browser must be configured to only use the proxy for requests to the ASP that provided the proxy, either by the use of a proxy auto-config file[2] or manually configuring the browser to do this. Once the proxy is initialised, it retrieves a SLA for the customer from the ASP via HTTP, which configures the proxy to monitor specific performance and availability parameters of requests to the service provided by the ASP. The proxy may analyse the information gathered locally before sending it to the Service Provider over HTTP or may just send all the logged information with no analysis being done locally. The Service Provider when processes the data received against the customers SLA which it already has and verifies that the service is SLA compliant. The Service Provider can then make a report available to the customer on service compliance. A block diagram of this architecture is given next showing the interaction between the Customer and Service Provider.

[2] A proxy auto-config file allows the automatic configuration of HTTP proxies so that optimal use is made of network resources. It supports features such automatic redirection to a secondary proxy if a primary one fails, load balancing between a number of proxies and the use of specific proxies for certain resources.

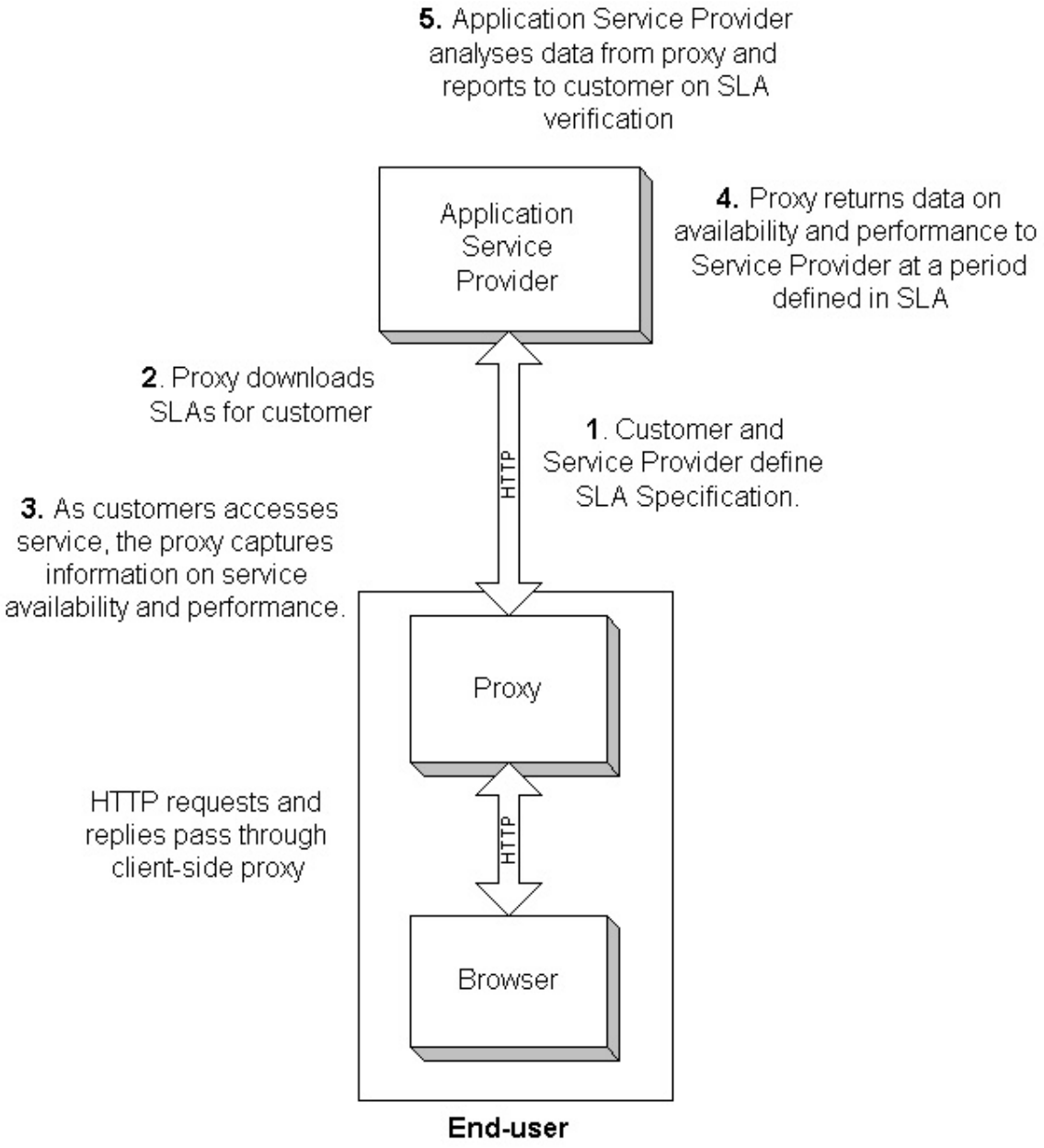


Figure 6: Service monitoring between Customer and a single Service Provider

This architecture will work for the scenario described. However it presents a number of issues.

- The proxy only monitors access to a single Service Provider at a time.

- If other ASPs are required to provide the same type of SLA monitoring and verification, an additional proxy will have to be deployed and configured by the customer for each ASP adding to management and processing overhead.
- Provides no way to compare performance of services provided by different ASPs.

Leading on from this architecture, the idea of SLA verification as a service will be considered. In this scenario, the customer or the ASP may approach a third party to verify that the service is SLA compliant. This has a number of advantages.

- Only one proxy needs to be deployed to monitor multiple services from multiple ASPs. In this case, the party that delivers the SLA Verification Service will provide the proxy to the customer
- The Customer may prefer having an independent third party verify SLA compliance.
- The SLA Verification Service Provider (SLAVSP) can monitor multiple service from multiple providers and possibly provide information to the ASP that provides each service.

To enable a third party to provide this service, it needs access to the SLA for the Customer and performance and available data captured by the proxy that the customer uses. Clearly co-operation is required between all parties involved in the service including the Customer, ASP and the SLAVSP

The ASP will still want to know how the service is performing for its Customers. However, the ASP now has the option of taking the data collected by the proxy and analysing it directly, or a more flexible approach would be for the SLAVSP to report on SLA Compliance. Since the proxy can do a certain amount an analysis locally, a mechanism may also be introduced whereby once the proxy detects that the service is not performing to specification, it alerts the ASP. The ASP may then retrieve information from the SLAVSP to diagnose the problem.

4.2.2 Multiple ASPs and Multiple Services

The difficulties that were highlighted with the previous model have resulted in the following architecture.

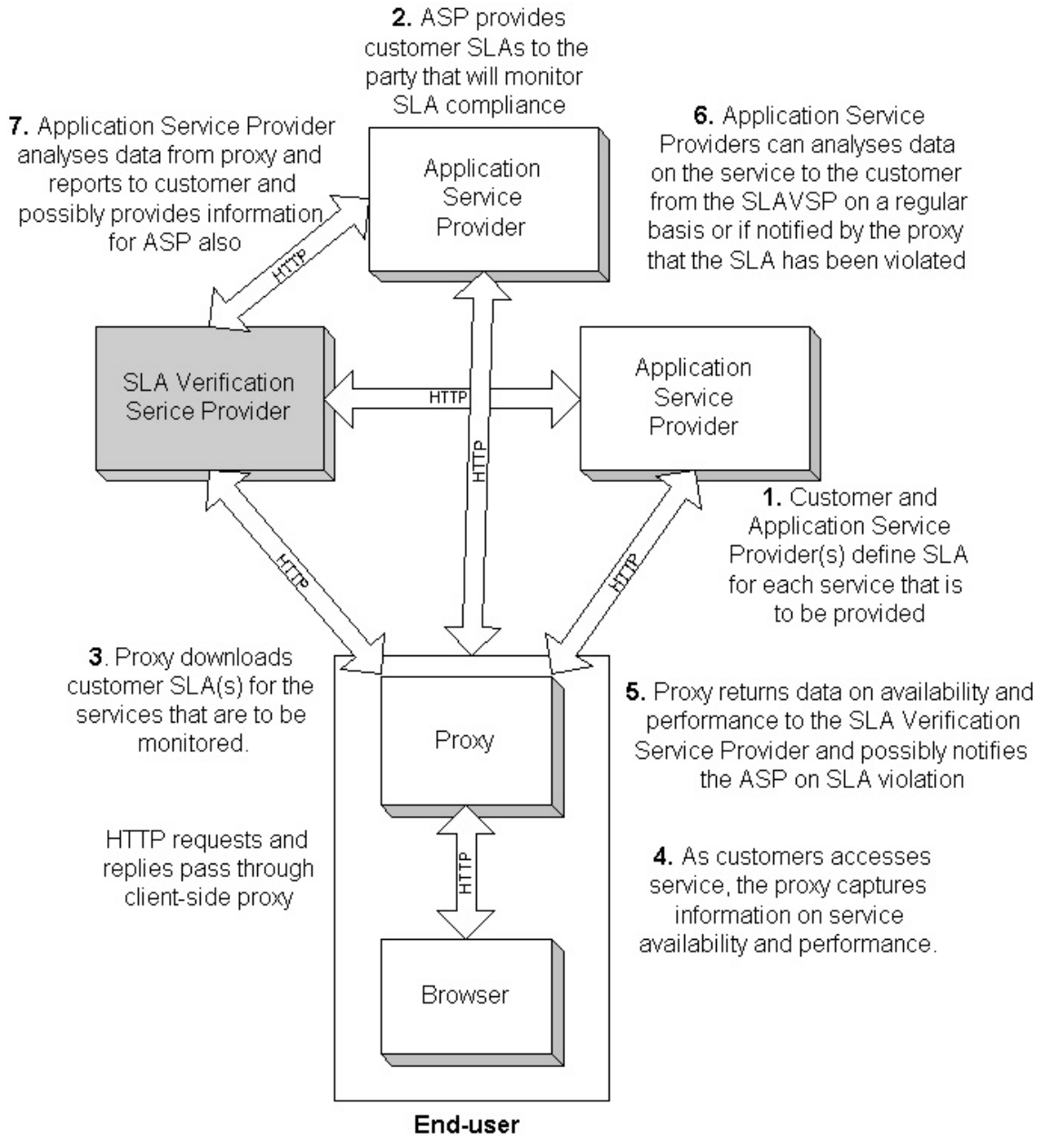


Figure 7: Service Monitoring between Customer and a Multiple Service Providers

4.3 Architecture in Operation

The operational description of the architecture will now be given:

1. The customer approaches the ASP to provide a service that will be delivered via the Web. They negotiate an appropriate SLA for performance and availability of the service.
2. The Service Provider makes the SLA(s) that have been defined with the Customer available to the SLAVSP that the customer will use monitor the service. The SLA can be made available via HTTP or some other mechanism for the SLAVSP to access it.
3. The SLAVSP adds the SLA to a list of monitored services for the customer.
4. The proxy on the client is configured to contact the SLAVSP when it starts up. It identifies the customer that is using the proxy to the SLAVSP and is returned a list of services to monitor. The DTD to describe the structure of this data is defined in Appendix B.
5. Once the proxy detects that a new service requires monitoring, it automatically retrieves the appropriate SLA for that service and configures the proxy to log information for the service described in the SLA. The DTD to describe the structure of this SLA data is defined in Appendix A.
6. The proxy will retrieve the list of services from the SLAVSP at a period that is configurable. If the proxy cannot contact the SLAVSP, it will attempt to reconnect to the SLAVSP at a predefined period.
7. The proxy then retrieves a SLA for each service that was listed.
8. The proxy analyses the SLA and is immediately configured to log information related to the service.
9. Periodically the proxy will analyse the data logged for each service and determine if SLA compliance is achieved. The frequency and period of service usage that the analysis runs for is determined by parameters contained in the SLA. The data is then summarised and sent to the SLAVSP via HTTP.
10. If the analysis determines that a SLA violation occurred, the proxy may notify the ASP. The ASP can then retrieve the data that the proxy passed to the SLAVSP and use the information to diagnose the service being delivered to the customer.

4.3.1 Additions and Modifications to Services Monitored

The ability to make modifications to the Services that are monitored for a Customer operates as follows:

1. Each time the list of services it retrieved by the proxy from the SLAVSP, it checks for any modification since the last retrieval.
2. If the proxy detects a new service identified, which uniquely identified the Service to the Customer, a SLA for that Service is retrieved from the SLAVSP. The service list will contain the necessary information that allows the correct SLA to be retrieved.
3. If the proxy detects that an update to the SLA for a Service has occurred, a new SLA for that service is retrieved from the SLAVSP. This is possible as each version of the SLA for a particular service has a unique contract identifier.
4. If the contract identifier returned in the list of service is set to –1 for a particular service identifier, the service is no longer monitored.

This architecture overcomes the inflexibility that the first architecture was prone to. What has been highlighted in the previous section is that a common method to describe Service performance and availability as perceived by the end-user is necessary. Considering that the SLAVSP may be required to monitor multiple Services delivered by multiple ASPs for multiple Customers, a generic way to describe Services is of the utmost importance.

4.4 Client-Side Proxy Design and Operation

Having now defined a flexible architecture that can support monitoring of Web delivered services, the design of the client-side proxy that will capture the necessary data is described in the following section including the process of how information on a request and its response is captured.

4.4.1 Proxy Architecture Overview

The client-side proxy that is used to analyse HTTP requests is based on Jigsaw. Jigsaw is the W3C reference server. Its main purpose is to demonstrate new protocol features as they are defined (such as HTTP/1.1), and to provide the basis for experimentations in the field of server software [20].

Jigsaw is written in Java to take advantage of threads and garbage collection, which allows for very dynamic and flexible server architecture. The primary reason that Jigsaw was chosen for the client-side proxy architecture is that it is highly extensible and allows what are referred to as Frames, which control how Resources (objects exported and made accessible to the outside world) are served using a specific protocol. It also allows what are referred to as filters, to be attached to the Protocol Frame (such as a HTTPFrame) of a Resource. A Filter is a full Java Object, associated to a Frame, which can modify the Request and/or the Reply. Usually filters are called before and after serving a Resource.

One of the frames that Jigsaw provides is a ProxyFrame. The Jigsaw ProxyFrame implements a full-blown proxy module for Jigsaw so that it can be configured to run as an HTTP proxy. It relies on the W3C's HTTP client side API to handle both request forwarding and reply caching on the end-users behalf. The client side API emulates the interactions with the origin server that the end-user’s browser would normally make. The HTTPD component of Jigsaw accepts the requests from and sends replies to the end-user’s browser.

Filters in Jigsaw can be classified in two types, server-side and the client-side filters. Server-side filters are applied to requests to the whole server if it is running as a Web server or proxy. A client side filter applies to the use of W3C's HTTP client side API in Jigsaw.

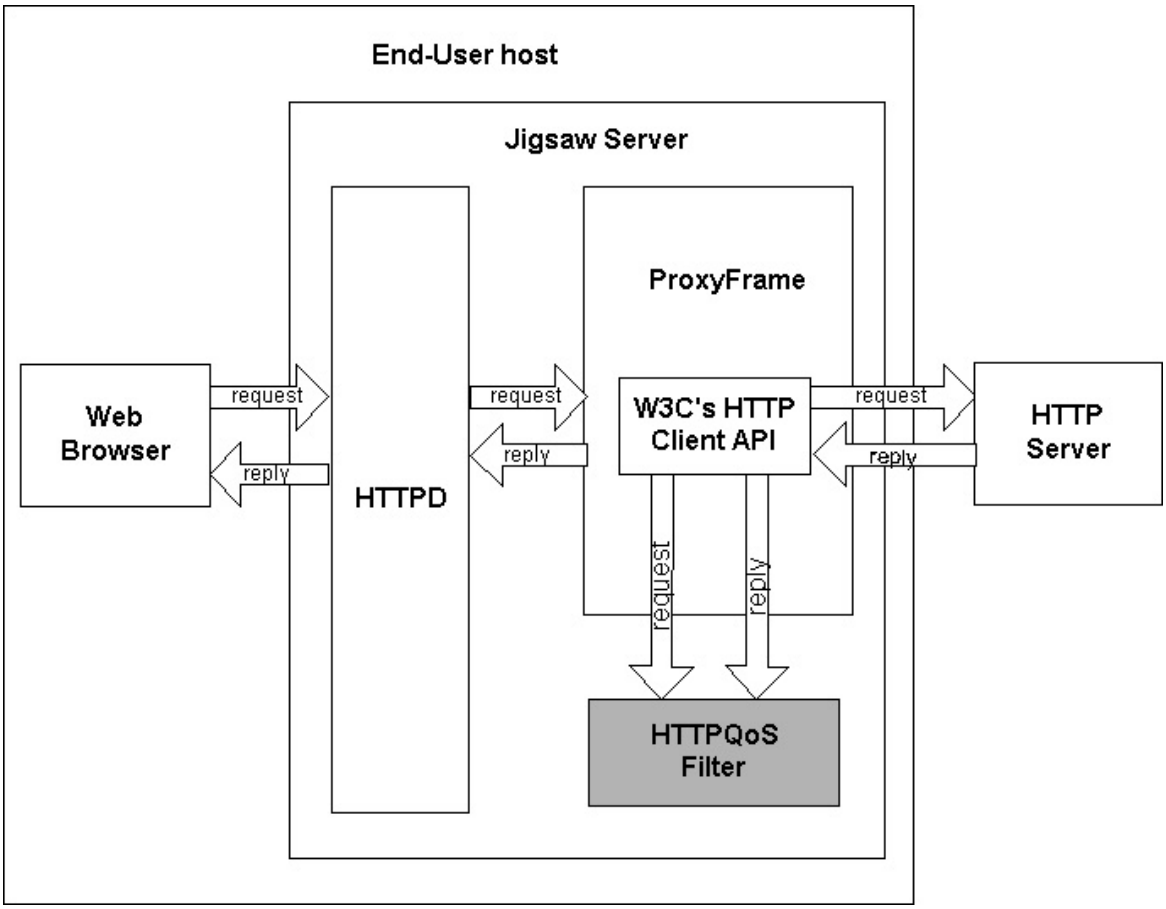


Figure 8: Jigsaw Proxy Configuration with HTTPQoS Filter

Typically the ProxyFrame is configured to run with a CacheFilter to enable caching or ICPFilter to allow the proxy to support the Internet Caching Protocol. ICP is a lightweight message format used for communication among Web caches [21]. However, the function of the proxy that was required in the situation here was to be able to monitor HTTP requests and replies so these filters were not enabled. A custom filter (HTTPQoS), shown in the previous diagram, was designed to enable the capturing of the appropriate parameters required as discussed in the Chapter 3 of this dissertation. The details of this filter will be described next and followed by its use in the overall proxy architecture.

4.4.2 HTTPQoS Filter Architecture

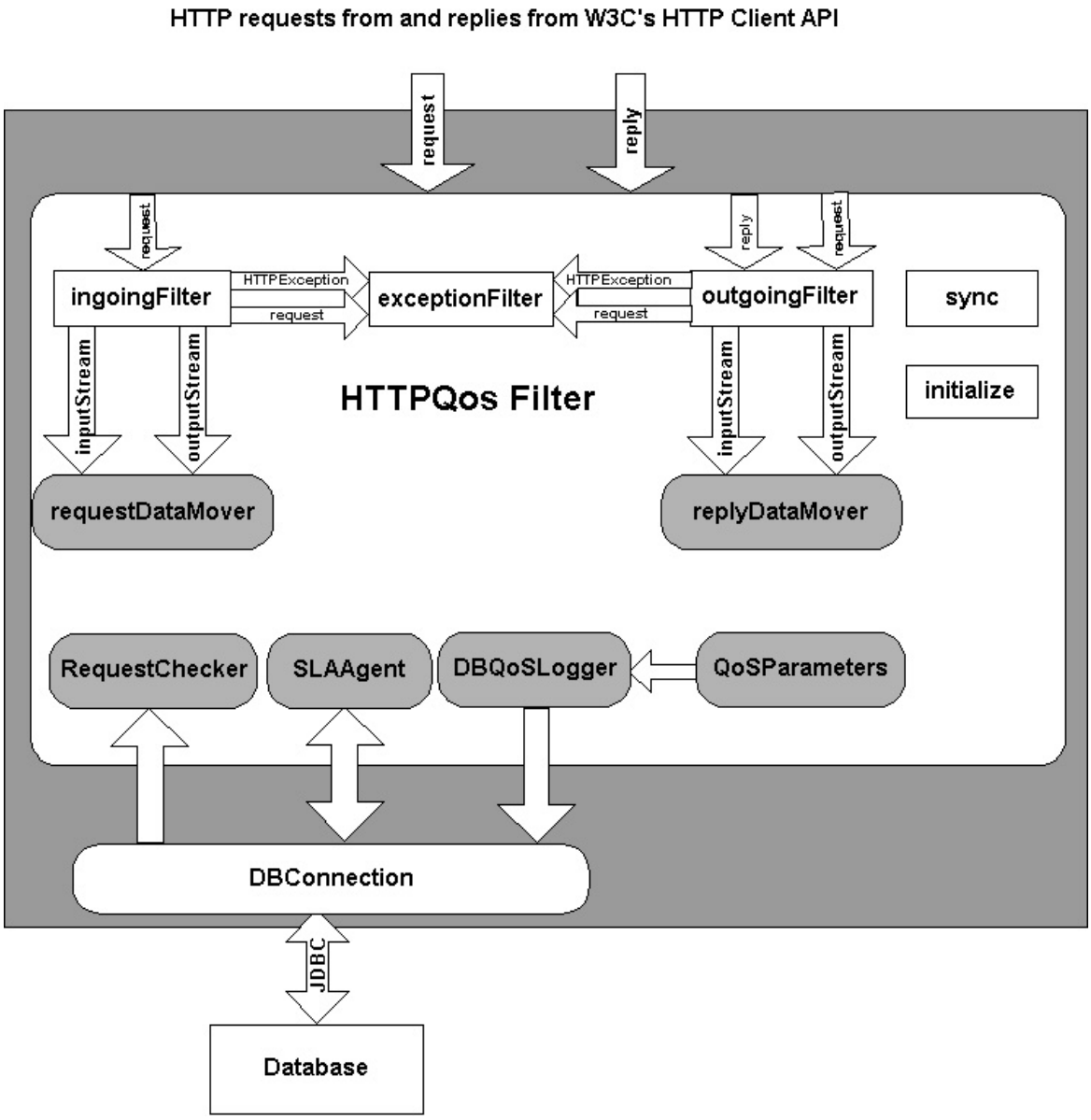


Figure 9: HTTPQoS Filter Architecture Overview

The HTTPQoS filter is applied to run against the client side HTTP API that Jigsaw uses to handle both request forwarding and reply caching. The *HTTPQoS* filter can catch requests before they leave the proxy and get replies as they come back from the origin server. This feature of the *HTTPQoS* filter is used whereby the *ingoingFilter* is called when the request is

sent to the server and the *outgoingFilter* is called when a reply is returned from the server. If an exception occurs in *ingoingFilter* or *outgoingFilter*, an exception is caught and the *exceptionFilter* is called. The use of *HTTPQoS* within Jigsaw for a typical HTTP request is new presented.

4.4.3 Operation of the Jigsaw Proxy and the HTTPQoS filter

The use of the Jigsaw proxy with the HTTPQoS filter occurs as follows for a typical HTTP request.

1. The end-user's browser is configured to pass all HTTP requests via the local proxy. When the end-user enters a URL in their web-browser, the request is passed to the *HTTPD*, which passed the request to the ProxyFrame.
2. The ProxyFrame then uses the HTTP client side API to make the request to the target server. Any requests that are passed to the client side API will use the *HTTPQoS* filter. For the request, the *ingoingFilter* method of *HTTPQoS* gets called and it's passed the full request object.
3. The request is passed to *RequestChecker*, which determines if the request needs to be monitored. *RequestChecker* examines the request's URL and method and if there is a mapping in any of the SLAs that corresponds to the request URL and method, the request should be monitored.
4. Assuming the request should be monitored, an instance of *QoSParameters* is created. A unique identifier for the request is generated and assigned to a request header called *requestID*. Various data relating to the request is inserted into the instance of *QoSParameters*.
5. The instance of *QoSParameters* is then inserted into a hash table indexed by the requestID so that the instance of *QoSParameters* can be referenced later when the reply is received.
6. If the request has an output stream associated with it, *RequestDataMover* is called and is passed the instance of *QoSParameters*. It then analyses the output stream of the request. *RequestDataMover* calculates the amount of data in the output stream and the time taken

to transfer the data to the server. This information is stored in the instance of *QoSParameters* that is passed to *RequestDataMover*.

7. If the request has no output stream, the ingoingFilter is exited.
8. The target server then handles the request and a reply is returned. The reply is passed back to the ProxyFrame via the client side API.
9. The *outgoingFilter* of *HTTPQoS* gets passed the original request and reply that has just been returned.
10. The request associated with the reply is passed to *RequestChecker* again to determine if the reply needs to be analysed.
11. If *RequestChecker* indicates that the reply should be analysed, an instance of *QoSParameters* is instantiated to the original *QoSParameters* instance that was set in the *ingoingFilter*. This is possible because the request was also passed to the *outgoingFilter*, and the value of the request header, requestID can be accessed from the headers of the request. This values is then used to retrieve the instance of *QoSParameters* from the hash table that was populated in ingoingFilter.
12. *QoSParameters* is populated various parameters such as the time that the reply returned and as the reply status code.
13. If the reply has an input stream associated with it, *ReplyDataMover* is called and is passed the instance of *QoSParameters*. *ReplyDataMover* then analyses the input stream of the reply and calculates the amount of data and the time taken to transfer the data from the target server. *QoSParameters* is then passed to *DBQoSLogger*, which inserts the information stored in the instance of *QoSParameters* into a local database via *DBConnection*.
14. If the reply has no input stream, *QoSParameters* is passed to *DBQoSLogger* and inserted into a local database via *DBConnection*.
15. The instance of *QoSParameters* is then removed from the hash table and the outgoingFilter is exited and the full reply is returned to the end-user via *HTTPD*
16. If an HTTP exception occurs in the *ingoingFilter* or *outgoingFilter*, the *exceptionFilter* get passed the request and HTTP exception that occurred. An instance of *QoSParameters* is instantiated with a reference to the original *QoSParameters* instance that was set in the *ingoingFilter*. This is possible because the request is also passed to the *exceptionFilter*,

and the value of the request header, requestID can be accessed from the headers of the request. This value is then used to retrieve the instance of *QoSParameters* from the hash table that was populated in *ingoingFilter*. The exception that occurred is inserted into *QoSParameters*. *QoSParameters* is then passed to *DBQoSLogger* and inserted into a local database via *DBConnection*. The instance of *QoSParameters* is then removed from the hash table and the *exceptionFilter* is exited and an error message is returned to the user via *HTTPD*

Having described the interaction of *HTTPQoS* with Jigsaw in detail, the purpose of the individual classes used by *HTTPQoS* is listed in the following table.

Class	Description
RequestChecker	RequestChecker provides information on SLA mappings for all services that the customer wants monitored. This information is used by *HTTPQoS* to determine if a particular request should be monitored.
SLAAgent	Retrieves the list of services from the SLAVSP for the Customer via HTTP. The information returned is in XML which is parses to determine if any service that is currently monitored has been updated or new services added since the previous retrieval. It logs the retrieved information to the database if appropriate. *SLAAgent* retrieves the service data at a specific period that's defined in the information downloaded. If the SLAVSP cannot be contacted, the SLAAgent will attempt a connection again in a predefined period.
QoSParameters	Stored information on each request that is to be monitored.
DBQoSLogger	Logs the data contained in an instance of *QoSParameters*
RequestDataMover	Analyses the output stream associated with the message body of a request. It calculates the amount of data contained in the message body and the time taken to upload the data to the target server.
ReplyDataMover	Analyses the input stream associated with the message body of a reply. It calculates the amount of data contained in the message body and the time taken to download the data from the target server.

Table 4: Table of Classes Used by HTTPQoS

4.5 Information Architecture

The overall information architecture of the framework will now be described. This includes the design of the SLA that can be used to describe the QoS requirements and other information as perceived by the end-user of a Web delivered Service.

4.5.1 SLA Design

The design of the SLA to describe a Web delivered service, as perceived by the end-user is a key component to the overall framework. The SLA needs to be able to capture information relating to performance, availability, monitoring and trouble handling aspects of a Service delivered via the Web in a generic way so that a SLA for any Web delivered service can be defined. The areas that are focused on in this dissertation are general items, QoS aspects, trouble handling, and monitoring and reporting aspects. The accounting or security aspects of SLAs are not addressed.

4.5.1.1 SLA Breakdown

A breakdown of how to represent the components in a SLA for a Web delivered Service will now be undertaken. Two main groups of parameters/subgroups are defined as follows.

Grouping	Description
General Parameters	Contains general items such as identifiers for the service, information relating to availability and performance requirements, action to take for trouble handling and monitoring and reporting aspects.
QoS Parameters	Contains specific QoS parameters for the service being delivered

Table 5: SLA Parameter Groupings

These groupings can contain sub groupings or just parameters.

Parameter/Parameter Grouping	Description
SERVICE-ID	Uniquely identifies the service to the customer
CONTRACT-ID	Uniquely identifies the SLA in conjunction with the SERVICE-ID
SERVICE-NAME	Parameter to indicate the name of the Service
SERVICE-AVAILABILITY	Parameter to indicate required service availability
SERVICE-PERFORMANCE	Parameter to indicate required service performance
SLA-VIOLATION-ACTION	Grouping of parameters that define the action to take if an SLA violation occurs
SLA-REPORTING-INFO	Grouping of parameters concerning SLA reporting
SERVICE-REQUESTS-PER-SECOND-LIMIT	Parameter to indicate number of requests per second to the service that the SLA can be applied to
SERVICE-ERROR-HEADER	Identifies the header to be returned by the service provider in the case of a backend component failure that results in a service failure but which returns a valid HTTP reply

Table 6: Table of General Parameter Groupings

Grouping	Description
REFERED-OBJECTS-PERFORMANCE-PARAMETERS	Grouping of parameters for requests to referred URLs
SERVICE-MAPPINGS	Grouping of service function parameters

Table 7: QoSParameter Groupings

Based on these two higher groupings, the overview of the structure of the SLA is as follows:

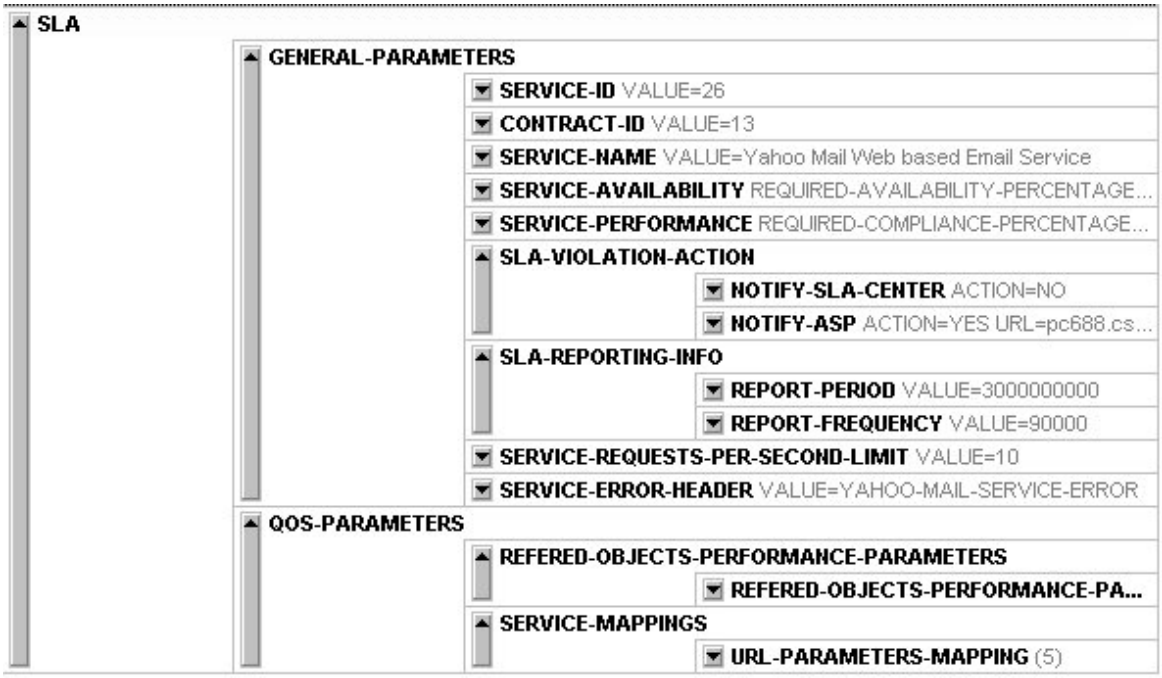


Figure 10: General parameter and QoS parameter groupings

4.5.1.2 Assigning Parameters to Service Functions in the SLA

The variations in the interactions with HTTP for different service functions, or what could be considered HTTP resources, require a considerable amount of flexibility in how parameters are assigned. This was highlighted in the previous chapter where it was demonstrated that some derived parameters are more applicable to certain service functions and a different number of parameters may want to be applied to a particular service function. Assuming that the following URL is requested in a Web delivered service to take the input of a HTML form using a POST method

http:// webmail.cs.tcd.ie/sendMail

First we assign this URL and the method associated with it to a service function name, in this case, we'll call it Send Mail

Mapping Name = *Send Mail*

- ***URL*** = *http://webmail.cs.tcd.ie/sendMail*
- ***Method*** = *POST*

Then the parameters that we wish to measure for this mapping name are specified
For example:

HEADER-RESPONSE-TIME 2000 MAX

This means that the parameter HEADER-RESPONSE-TIME should be a maximum value of 2000. There can be more than 1 parameter per mapping however and we may wish to specify the upload rate also as follows:

ENTITY-UPLOAD-RATE 1 MIN

This specifies that the rate that an entity is uploaded at should be a minimum of 1.

4.5.1.3 Describing Parameters Assigned to a Service Function using XML

Now lets representing the mapping above in XML:

```
<URL-PARAMETERS-MAPPING                    Start of Mapping
MAPPING-NAME="Send Mail"                   The mapping name
MAPPING-ID="1">                            The mapping identifier
        <REQUEST
        METHOD="POST"                      The request Method
        URL="http://webmail.cs.tcd.ie/sendMail"/>   The request URL

        <PERFORMANCE-PARAMETERS>           Start of Performance Parameters
                <PERFORMANCE-PARAMETER     Performance Parameter
                NAME="HEADER-RESPONSE-TIME"  Parameter Name
                VALUE="2000"               Parameter Value
                FUNCTION="MAX"/>           Function & end of Parameter
                <PERFORMANCE-PARAMETER     Performance Parameter
                NAME="ENTITY-UPLOAD-RATE"  Parameter Name
```

```
            VALUE="1"                                   Parameter Value
            FUNCTION="MIN"/>                            Function & end of Parameter
        </PERFORMANCE-PARAMETERS>                       End of Performance Parameters
        <MONITOR-REFERED-OBJECTS VALUE="NO"/>           Monitor Referred objects or not
</URL-PARAMETERS-MAPPING>                               End of Mapping
```

For any service there will usually be multiple mappings like the one described previously.

The complete DTD for a SLA is given in Appendix A.

4.5.2 Parameter Definition

The ability to derive parameters from the base parameters logged for each request was described in the Chapter 3. It was decided that for flexibility, the queries that calculated the derived parameters should be defined using XML and then stored in the local database. This has the advantage of making the framework flexible in that it is possible to define new parameters without having to modify the source-code of the framework.

4.5.3 Database Design

A number of components in the HTTPQoS filter make use of an Oracle Lite database to store and retrieve data. This includes the storage of data collected for HTTP requests and responses, SLAs that are downloaded from the SLAVSP and data on how to calculate derived parameters described previously.

Table Name	Description
QOSParameters	Used to store information relating to requests and responses to and from Services that are monitored
SLAGeneralParameters	Stores information of the general SLA parameter grouping
SLAServiceMappings	Stores the service mappings that are defined for each service
SLAServiceMappingParameter	Stores data on the derived parameters that are to be monitored for each service mapping
SLAReferedObjectsParameter	Stores parameters that are to be monitored for referred requests.
SLAParameterDesciptor	Stores the information of how derived parameters are calculated

Table 8: Database Tables and Descriptions

4.6 Conclusion

This chapter initially presented a number of architectures that could support QoS monitoring for Web delivered Service. The overall design of the client-side proxy was then discussed including the *HTTPQoS* filter that actually analyses HTTP requests and replies and logs appropriate performance and availability information to a database. The information architecture of the framework was featured next. This included the design of a generic SLA for Web delivered Services in XML and a number of DTDs that define other data structures to support the framework. Completing the Chapter, the database tables used to store and retrieve data were listed and their functions described.

5 IMPLEMENTATION

5.1 Introduction

This chapter discusses the implementation details of the framework. This includes the implementation of the *HTTPQoS* filter and the process of monitoring data streams to and from a HTTP server. Where appropriate, source code or pseudo-code will be used to explain how features were actually implemented. Finally, the configuration of the *HTTPQoS* filter on Jigsaw will be discussed.

5.2 HTTPQoS Filter

A key concern was the issue of the *HTTPQoS* filter implemented was that the filter should not have a significant impact on end-user perceived performance for requests that are passed through the proxy. The *HTTPQoS* filter is applied to run against the client side HTTP API that Jigsaw uses to handle both request forwarding and reply caching. The *HTTPQoS* filter can catch requests before they leave the proxy and get replies as they come back from the origin server. This feature of the *HTTPQoS* filter is used whereby the *ingoingFilter* is called when the request is sent to the server and the *outgoingFilter* is called when a reply is returned from the server. If an exception occurs in the *ingoingFilter* or *outgoingFilter*, an exception is caught and *exceptionFilter* is called.

The fact that different methods are called in the *HTTPQoS* filter when a request is sent or a reply received allows specific parameters to be logged such as the time that a request was made and the time that a reply was received. The *ingoingFilter* gets passed the full request object so it can access the header fields and the entity body associated with the request. The *ingoingFilter* makes a call to a separate object called *RequestDataMover* that recorded the rate that the input stream associated with the request was returned from the server and passed to the client

The *outgoingFilter* get passed the request and reply object. Again, a call to a separate object called *ReplyDataMover* is made that recorded the rate that the output stream associated with the reply was returned from the server and passed to the client
The *exceptionFilter* method get passed the request and HTTP Exception that resulted in it being called so the *exceptionFilter* that can catch any exceptions that occur in the processing of a request or reply.

A number of obstacles had to be overcome in the design of the *HTTPQoS* filter so that the filter did not impact performance. These will now be discussed.

5.2.1 Monitoring HTTP Message Content Length and Transfer Time

To monitor the performance of a Web delivered service, determining the rate of data transferred from the end-user to the target HTTP server and vise-versa needs to be calculated for both http requests and replies. The amount of data in a HTTP message body can usually be determined from the *Content-length* header field. However, sometimes this field is not propagated with information. For example, with chunked transfer-coding, the *Content-length* header field is not used so some other method has to be used to determine the information.

The time that the data transfer of the message body starts and finished needs to be recorded also to determine how long the transfer takes. To add to the complexity of this, it has to be done in such a way so as not to impact performance. Two separate classes are use by HTTPQoS filter to enable such monitoring to take place. These are *RequestDataMover* and *ReplyDataMover* and they monitor data transfer for HTTP requests and replies to and from a target server respectively. The detail of how *RequestDataMover* is used by *HTTPQoS* is worth examining as it highlights the challenges that monitoring performance present and how *HTTPQoS* overcomes impacting performance.

The code below is taken from *ingoingFilter* in *HTTPQoS*

Line 1 detects if the request has in input stream to measure. Line 3 creates an instance of PipedOutputStream named *pout*. Line 4 creates an instance of *PipedInputStream*, *pin* to which *pout* is passed as a parameter. This effectively means that whatever is fed to *pout* will automatically be passed to *pin*.

HTTPQoS.java

```
1. if(request.hasOutputStream()){
2.    try {
3.       PipedOutputStream pout = new PipedOutputStream();
4.       PipedInputStream  pin  = new PipedInputStream(pout);
5.       new RequestDataMover(request.getOutputStream(),pout,myQOSParameters);
6.       request.setOutputStream(pin);
7.    }
8.    catch (Exception ex) {}
9.    }
10. }
```

In line 5 a new instance of *RequestDataMover* in instantiated, the constructor of which is passed the current requests input stream, the instance of PipedOutputStream *pout* and an instance of *QoSParameters*. Finally in line 6, the requests output stream is passed *pin*. What this effectively allows is to monitor the request's output stream without interrupting it as it is passed to the target server. The implication of this are that the time data flows can be measured as can the amount of data transferred to the server. Additionally since the *RequestDataMover* is threaded it does not cause the filter to block.

Now lets look at what happens in *RequestDataMover*.

Once the constructor of *RequestDataMover* is called, the references to the current requests input stream, the instance of PipedOutputStream *pout* and an instance of *QoSParameters* are assigned to global variables in *RequestDataMover*. Then *start()* is called which has the effect of calling *run()* which is examined further on the next page.

RequestDataMover.java

```
1. class RequestDataMover extends Thread {
2.     InputStream in = null;
3.     OutputStream out = null;
4.     QoSParameters myQoSParameters;

5.    RequestDataMover(InputStream in, OutputStream out,QoSParameters qoSParameters)
6.    {
7.        this.myQoSParameters = qoSParameters;
8.        this.in  = in;
9.        this.out = out;
10.       setName("RequestDataMover");
11.       start();
12.    }
```

In *run()*, first the time is recorded indicating the start time of the data stream(line 19). The input stream, *in*, is read at a byte per iteration, (line 20). This data is when written to *out* (line 22). The amount of read data is tracked also (line 23). When there is no more data to read, the end time is recorded (line 25), as is the amount of data that has been transferred (line 26).

Examining this closer, *out* in *RequestDataMover* is a reference to *pout* in *HTTPQoS* and *pout* is fed to *pin* in *HTTPQoS*. Since the request's output stream in *HTTPQoS* is passed *pin*, this allows the data can be monitored effectively while transferring it to the server but not interrupting the data stream.

RequestDataMover.java ..continued

```
13.
14.   public void run() {
15.      try {
16.         byte buf[] = new byte[1];
17.         int  got   = -1;
18.         myQoSParameters.setRequestBodyStartTime(System.currentTimeMillis());
19.         int contentLenght = 0;
20.         while ((got = in.read(buf)) >= 0){
21.            out.write(buf, 0, got);
22.            contentLenght = contentLenght + 1;
23.         }
24.         myQoSParameters.setRequestBodyEndTime(System.currentTimeMillis());
25.         myQoSParameters.setRequestContentLength(contentLenght);
26.      } catch (IOException ex) {
27.         ex.printStackTrace();
28.      } finally {
29.   try { in.close(); } catch (Exception ex) {};
30.   try { out.close(); } catch (Exception ex) {} ;
31.   }
32.}
```

ReplyDataMover similarly does the same the process for a request in HTTPQoS except data is passed in the opposite direction. The effectiveness of the features described previously to

prevent blocking or data stream interruption will be determined in Chapter 6, when the performance impact of the HTTPQoS filter is analysed.

5.3 DBQoSLogger

Since the action of logging parameters to the database has the potential to become a bottleneck and impact performance, it was decided that the logging should be a non-blocking implementation. To log an instance of *QOSParameters*, an instance of the class *DBQoSLogger* is created whose constructor is passed the instance of *QoSParameters* as follows.

HTTPQoS.java

```
1.  new DBQoSLogger(QoSParameters);
```

DBQoSLogger extends the Tread class in Java, a separate thread of the class *DBQoSLogger* now handles the logging to the database, so blocking will no longer occur.

5.4 QoSParameters

The *QoSParameters* class is used to store information in the HTTPQoS filter as requests and replies are made. For example in *ingoingFilter* of *HTTPQoS*, QoSParameters is used to store information such as the time that the request is made, the requestID and various other parameters that are used for monitoring. Once the information contained in *QoSParameters* needs to be logged, it is passed to *DBQoSLogger*, as described above.

HTTPQoS.java

```
1.  QoSParameters myQOSParameters = new QoSParameters();
2.  MyQOSParameters.setRequestTime(System.currentTimeMillis());
3.  MyQOSParameters.setRequestID(this.getRequestID());
```

5.5 Configuring the HTTPQoS Filter on Jigsaw

Once the *HTTPQoS* filter was ready to be configured on the proxy, the JigAdmin tool, which is the administration tool for Jigsaw, was started up.

The diagram below demonstrated the easy of configuration of the Jigsaw server. Here the HTTPQoS filter is applied to ProxyProp, which is the Proxy Properties controller for Jigsaw. Once this is configured, the Jigsaw will use the filter for requests and replies that are passed through it.

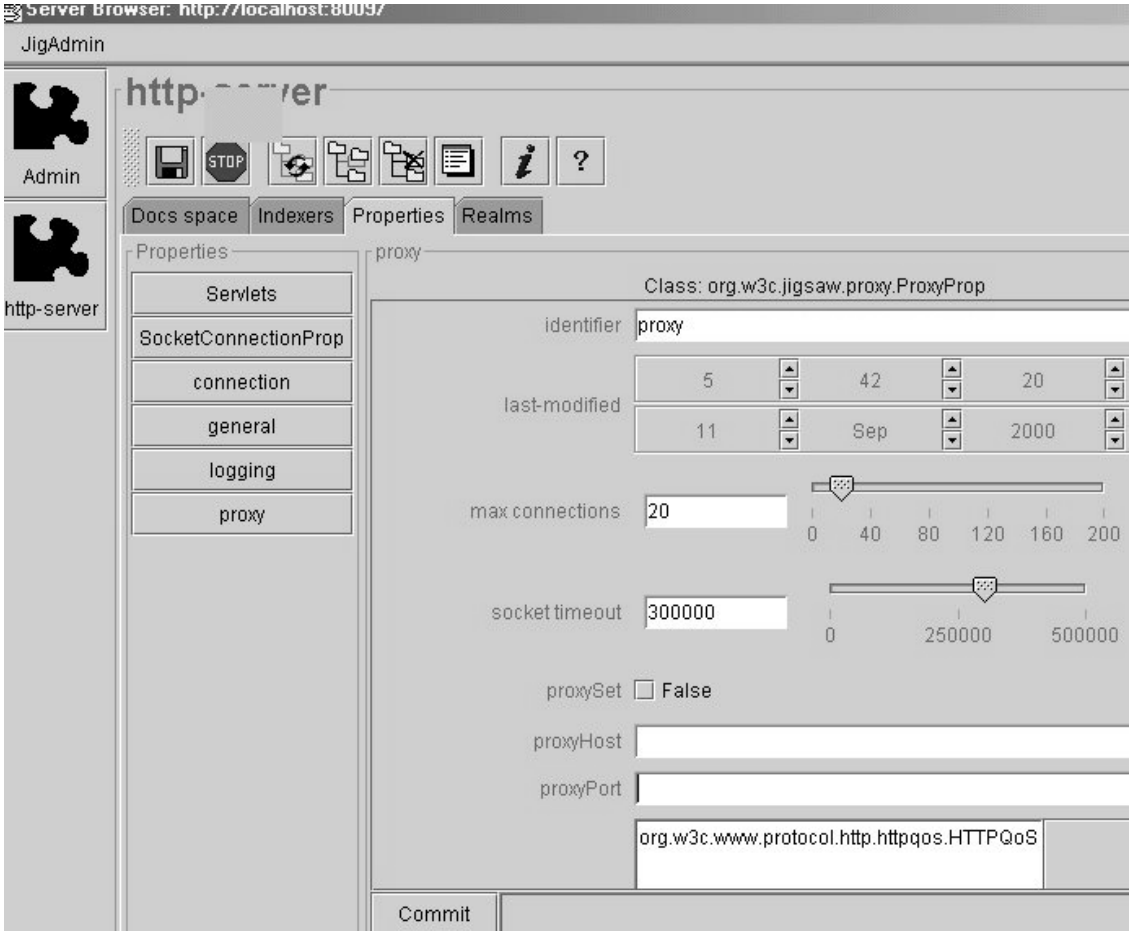


Figure 11: HTTPQoS Filter Configuration using the JigAdmin Tool

5.6 Summary

This chapter primarily discussed the implementation details of how HTTPQoS filter monitors data streams to and from a HTTP server. Finally the process of configuring the HTTPQoS filter on Jigsaw was covered briefly.

6 EVALUATION AND CONCLUSION

6.1 Introduction

The objective is this chapter is to evaluate the design and flexibility of the overall framework and the usefulness of the information that it can provide. Following this, a conclusion of the completed dissertation will be given and possible future work that could be undertaken to improve and extend the framework.

6.2 Evaluation Overview

This evaluation first examines a case study that was conducted using the framework to monitor a Web delivered email service. The aim of this was to demonstrate how the framework could be applied to monitor a Web delivered service and provide useful information on availability and performance from the user-perspective.

6.3 Case Study: Yahoo Mail

It was decided that to properly evaluate the framework, a demonstration of its capabilities should be undertaken against an actual Web delivered service. The target service that was chosen was Yahoo Mail. Yahoo Mail is a free Web based email service. It allows its subscribers to receive and send email via the Internet using a standard Web browser as an interface to the service. It supports all the traditional features that typical mail user-agents possess including the capabilities to send, receive, attach files to messages, search for messages etc, except the interface is a normal Web browser.

6.3.1 Test Environment

The test environment that was used to demonstrate the service consisted of a host with a Web browser and the Jigsaw proxy running with the HTTPQoS filter enabled. The Web browser was configured to pass all requests via the proxy. The Internet connection was via a dialup-account to a local ISP over a 56kbps modem. It was expected that the performance of

this connection would exhibit some variations in the responses and perhaps connection failures would occur which would highlight the monitoring capabilities of the proxy. Of course, dropping the Internet connection could simulate failures connecting to the service if required to generate appropriate logs.

The services list and the SLA for the service, which will be discussed next, were placed on a Web server running on the same host that the proxy runs on. This would not be the typical scenario in that the proxy would normally be configured to connect to the SLAVSP, identify the customer and be returned the appropriate list of services to manage. For each service, the appropriate SLA would then be retrieved if not already stored locally. However it was sufficient in this case to use the local Web server and static XML files to store the services list and SLAs so as to demonstrate the operation of the framework.

6.3.2 SLA Defined for Case Study

For the definition of the SLA, if was first necessary to examine how Yahoo Mail actually implements the delivery of the service using HTTP. The typical features that are offered in the service needed to be assigned to service functions. The service functions created for the case study were as follows:

6.3.2.1

Service Function	**URL**	**Method**
Login to Mail	http://login.yahoo.com/config/login?	POST
Read Mail	http://us.f1.mail.yahoo.com/ym/ShowLetter?	GET
Compose Mail	http://us.f1.mail.yahoo.com/ym/Compose?	GET
Delete Mail	http://us.f1.mail.yahoo.com/ym/ShowFolder?	POST
Send Mail	http://us.f1.mail.yahoo.com/ym/Compose?	POST
Search Mail Form	http://us.f1.mail.yahoo.com/ym/Search?	GET
Search Mail Results	http://us.f1.mail.yahoo.com/ym/Search?	POST
Logout of Mail	http://us.f1.mail.yahoo.com/ym/Logout?	GET
Attach to Mail	http://us.f1.mail.yahoo.com/ym/Attachments?	GET

Table 9: Yahoo Mail Service Mappings

It is worth noting that the same URL and method can provide the different service functions. The implication of this will be discussed later.

For the next step, parameters were assigned to each service mapping. For the purpose of the case study, three derived performance parameters were created to monitor which consisted of the following.

HEADER-RESPONSE-TIME	Response time in milliseconds to receive a reply header from time that request is sent
ENTITY-DOWNLOAD-RATE	Transfer rate in Kbps of reply body from service
ENTITY-UPLOAD-RATE	Transfer rate in Kbps of request body to service

Table 10: Table of Derived Parameters for Case Study

For the service mappings that were defined previously, parameter values were assigned. For example, *Read Mail* had the following performance values specified:

Parameter	Value	Function
HEADER-RESPONSE-TIME	2000	MAX
ENTITY-DOWNLOAD-RATE	2	MIN

Table 11: Parameters Assigned for Read Mail Service Mapping

This states that the header response time for a request to the *Read Mail* service function should be a maximum of 2000 and entity download rate of the message body greater that 2.

It is worth observing that these assigned parameters are appropriate for the *Read Mail* service function but may not be appropriate for other service functions. For example, assigning HEADER-RESPONSE-TIME to the *Send Mail* service function may not really provide any useful information. This is because if the request has a large amount of data to transfer it will impact the time that the full request is received by the server which will in turn impact the time that the reply is returned to the client. However it is possible to define a new derived parameter that could be used for service functions that use the POST method. For example, it could be possible to define HEADER-RESPONSE-TIME-AFTER-REQUEST-BODY which measures the time from when a request body stream ends and the time the reply header returns. Again this highlights the flexibility of the framework in that new derived parameters can be defined from the base parameters when needed. The list of base parameters is defined in Appendix F for reference.

For the Yahoo Mail case study, the required availability and performance parameters were defined as follows:

Minimum Service Availability Percentage Required	98%
Minimum Service Performance Percentage Required	95%

Table 12: Table of Required Availability and Performance for Case Study

Looking at how the SLA is defined, each SLA has a service identifier and contract identifier. The combination of the service identifier and contract identifier uniquely identifies the service to the proxy. The service identifier and contract identifier are defined at the start of each SLA as follows.

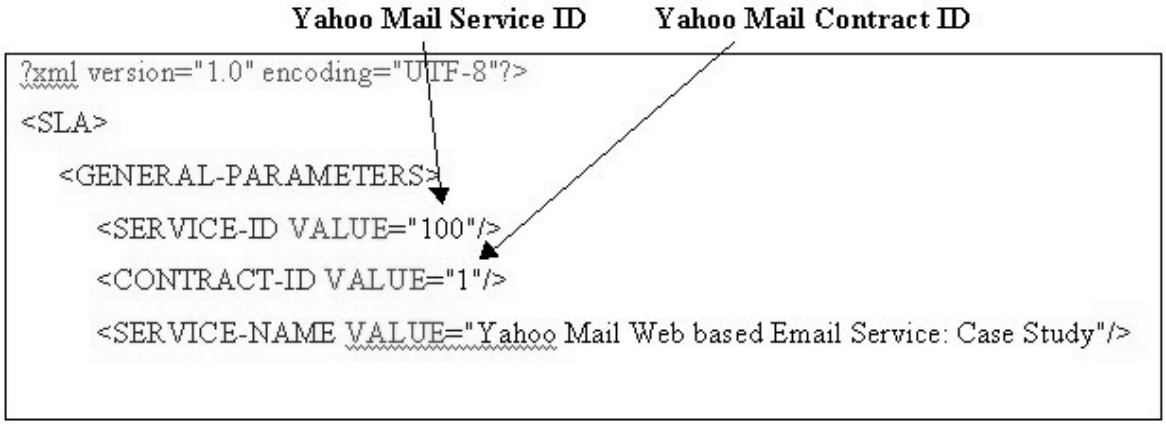


Figure 12: Service ID and Contract ID in SLA

Various other parameters were defined in the SLA but are not be mentioned here due to the fact that there are too many to list. However the actual SLA created is available in Appendix C for reference. The SLA created is based on the DTD defined in Appendix A.

6.3.3 Services List

Having defined the SLA, a services list for the proxy to download was created. The service files indicates to the proxy what services it should retrieve SLAs for. For the case study, the only service listed was the Yahoo Mail service, which was assigned a service ID of 100 and a contract ID of 1.

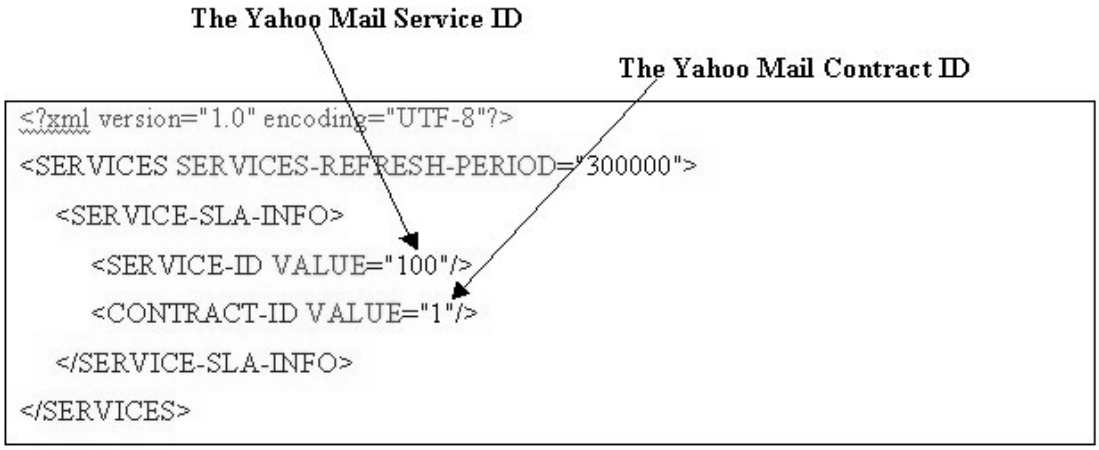


Table 13: ***Service File Specifying the Yahoo Mail service***

6.3.4 SLA and Services List Deployment

Once the SLA and services list was created, they were placed in a subdirectory of the Web server. The respective files were named *100-1.xml* and *services.xml*. First the proxy retrieves the services file and parses the xml to determine what services it should have SLAs for. For each service listed that the proxy does not have an SLA for already, it will retrieve a file with a naming convention of ***serviceID-contractID***.xml, serviceID and contractID being extracted from the service and contract identifier in the service list.

6.3.5 Proxy Initialisation

The proxy was configured to connect to the local Web server for the services list and SLAs. Once the proxy was initialised it connected to the Web server and downloaded the services

file. It then retrieved the appropriate SLA files, in this case 100-1.xml was the only one that had to be retrieved. This file then configured to proxy to log requests to Yahoo Mail.

6.3.6 Service Monitoring

For the purpose of the test a series of requests were made to the Yahoo Mail service over the period of one hour. The intention was to use the service as one would normally do but to try to access all the service mappings specified in the SLA so as to generate data on usage.

6.3.7 Results of Case Study

For this case study, a specific class named *SLAReporter* was designed to demonstrate the capabilities of the framework. It analysed the data logged by the proxy and the contents of the SLA for the service to determine if overall availability and performance are met. It then generated a HTML report containing the results of the analysis of the case study. The contents of the report for the case study are contained in Appendix E.

6.3.8 Report Analysis

SLAReporter was configured to run after an hour of service usage to Yahoo Mail. Based on the SLA and the data logged for the period of service usage, a report was generated. The report contained in Appendix E provided a breakdown of a number of different aspects of the service which include:

- Service Usage
- Service Performance
- Service Availability

The report provides overall performance and availability of the service compared with what was specified in the SLA. The service usage is also broken down by each mapping to indicating which are most frequently accessed service functions. Detailed performance information for each parameter is given including the minimum, average and maximum values observed. Then the overall availability of the service is reported and where failures

occur, the type of error is indicated. Finally, the service availability is broken down for each mapping specified in the SLA.

6.4 Strengths and Weaknesses of Framework

Having previously highlighted the usefulness of the information that the framework can capture, the strengths and weaknesses of the framework will now be discussed.

If one looks at the information that the report generated, monitoring at the server-side could capture some of this information. For example, service usage statistics can be captured by the Service Provider, as can HTTP server errors. There are a number of reasons why the framework developed offers a significant benefit over server-side monitoring.

- Server-side monitoring cannot detect connection attempts to a service that fail due to network related problems. With client side monitoring, these failed connections are possible to detect and record.
- The level of detailed monitoring that is implemented by the proxy may be a significant overhead to do on the server side for all customers. It may also be unnecessary, as particular customers may not require such monitoring. The framework developed allows detailed monitoring for selected Customers.
- The Service Provider may not wish to implement monitoring for a small number of customers, or may not have the expertise to do so. This framework provides a means for the Service Provider to offer SLA monitoring, verification and feedback via a Third party (SLAVSP).
- The fact that a Third party can be approached to verify SLA compliance may be preferable to the Customer than depending on the Service Provider.

Beside the advantages that the framework offers over server-side monitoring, it is also a more flexible approach over other possible methods of client side monitoring.

- If a Service Provider were to offer client side monitoring of service performance and availability, a specific proxy would have to be installed by the Customer for

every other Service Provider that offered the service also. The framework presented here allows a single proxy to monitor multiple services provided by multiple ASPs

- Comparing the performance of different Service Providers with a common methodology is possible with this framework. This would be useful to determine which Service Provider performs better from the customer perspective.

The benefits of the framework developed have now been highlighted. It is apparent that it offers greater levels of flexibility that server-side monitoring by the Service Provider or other methods of client side monitoring for SLA Verification.

The use of the framework during the case study raised one significant issue.

- The method of how mappings are defined in SLAs is not flexible enough. For the prototype implementation, the comparison that the *RequestChecker* class does of request URLs against mappings was simply a start of string comparison of the request against all the service mappings for the Customers. This may present problems as some Service Providers pass parameters via the POST method that indicate to the target service what service function is being accessed. Since these parameters do not appear in the query section of the URL, a string comparison cannot be done to determine the service function the user is accessing.

Overall the framework offers a flexible architecture for monitoring Web delivered services from the user-perspective.

6.5 CONCLUSION

6.5.1 Achievements

A final review of the objectives of the dissertation and what was actually achieved will now be examined. The primary objective of this dissertation was to develop a framework that

could support QoS monitoring and Service Level Agreements for Web delivered Services. This objective had three separate parts defined as follows:

- To design a client side proxy that supports automatic configuration, monitoring and logging of availability and performance parameters as perceived by the end user of Web delivered services
- The design of a SLA template specifically for Web delivered services that can map these parameters captured by the proxy in a meaningful way so that it is possible to specify how a service delivered over HTTP should perform, from the end-user perspective
- To implement SLA verification and feedback to the Service Provider and Customer so as to notify them of SLA compliance.

An overall architecture had to be designed for each of the previous parts to work together. This architecture was discussed in Chapter 4 and is shown in the next diagram

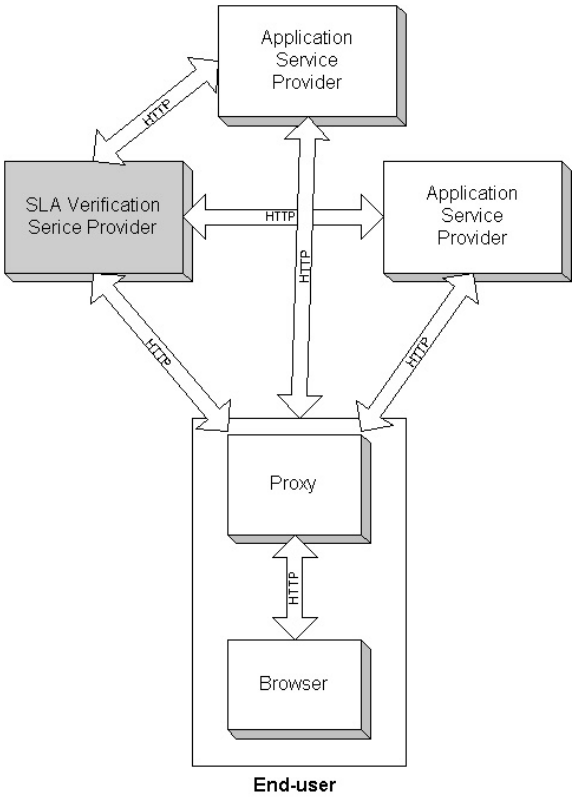


Figure 13: Framework Architecture

Once this architecture was decided on, the first part of the objective consisted of developing a proxy that could capture the necessary parameters to be able to monitor Web delivered services from the end-user perspective. This part of the framework was successfully implemented and provides a valuable piece of work that can be extended in the future. It must be mentioned that the extensibility of the Jigsaw server facilitated the development of this part of the framework immensely.

The next part was to develop a SLA template that could encompass the QoS requirements of a Web delivered service as perceived by the end-user. This necessitated examining the types of parameters that required monitoring for Web delivered services. There was a certain

amount of linkage between the SLA design and the capabilities of the client-side proxy as it was necessary to be able to capture specific parameters if they were to be specified in the SLA. The use of derived parameters made the SLA design a lot more flexible as it allowed parameters to be defined from the base parameters that the proxy captures. Again this part of the framework was successfully completed.

The third and final part of the primary objective was to investigate SLA verification and feedback to the customer and Service Provider. This part of the primary objective was only partially implemented due to time restrictions. The *SLAReporter* class implemented a basic verification and feedback system for the customer, which was discussed previously in the case study.

6.5.2 Future Development and System Improvements

The work completed in this dissertation was a proof-of-concept. It investigated the possibility of capturing performance and availability information of Web delivered services from the end-user perspective using a client-side proxy. It also investigated how to specify SLAs for Web delivered services that could encompass the QoS requirements as perceived by the end-user. These components became part of an overall framework that enable a Service Provider and the Customer to specify the performance and availability requirements of a service using a generic SLA. Once the SLA is defined it can be passed to a third party, which provides SLA verification and feedback to the Service provider and the Customer.

These features were successfully implemented but improvements and extensions could be made to a number of aspects of the framework. These include:

- The issue of what information to send data back to the SLAVSP and Service Provider needs further investigation including what processing should be done locally by the proxy and what should be done remotely.
- A more flexible method is needed that the current implementation to specify the mappings of service functions. The basic URL matching that was implemented in the

prototype could be improved and made more flexible than the current URL sub-string matching

- The use of a database to store the SLAs and other Xml structured data may have been inappropriate and perhaps inefficient as it necessitated parsing various XML structures and the inserting them into database tables. This information had then to be accessed later by using SQL. One option would be to store the services list and SLAs on the file-system of the host the proxy runs on instead of a local database. Then XQL could be used for data retrieval and filtering of the SLAs and services list. The bases parameters that are captured for each request and reply could also be written to a flat file in XML format and again XQL[1] could be use to calculate derived parameters.

6.5.3 Final Conclusions

The conclusions that can be drawn from the body of research completed in this dissertation are:

- The use of a client-side proxy for monitoring can be effective in capturing the performance and availability information of Web delivered services as perceived by the end-user.
- Client side monitoring offers the ability to detect problems that cannot be detected on the server side such as failed attempts to connect to the service, due to network problems for example.
- Client side monitoring offers a more scalable solution for monitoring services. The overhead of monitoring to such detail on the server side may be excessive and impact the service. It may also be unnecessary as not all customers may require such detailed monitoring.
- The framework suggested does not require significant configuration by the Service Provider to allow Customers to use a Third Party for SLA verification (SLAVSP). As

[1] The XQL Pattern syntax is simple yet provides powerful query capabilities. Its purpose is to identify a subset of an XML document based on ancestry chains, wildcards, and qualifiers such as attribute value tests. The syntax is concise and modeled after familiar methods for directory navigation. As a string-based syntax it can reside within attribute values, script languages, and URLs .[22]

such, the monitoring can take place independently of the Service Provider once the SLA has been given to the SLAVSP. Of course if the Service Provider requires feedback from the Customer, the necessary hooks will have to be put in place for the Service Provider to use.

- Being able to measure the QoS of a Web delivered service as perceived by the end user allows truly meaningful Service Level Agreements to be defined between the Service Provider and the Customer.
- The framework suggested in this dissertation is a flexible architecture that allows multiple services from multiple service providers to be monitor using the same proxy while also allowing feedback to the service provider via a SLAVSP.
- The location of where SLA verification and analysis are done can be distributed in the framework suggested, as the SLA **and** service usage data are available to the client side-proxy.

7 REFERENCES

[1] Microsoft, “Software as a Service: The Opportunity for Application Service Providers ” , 2000; http://www.microsoft.com/ISN/downloads/ASP%20partnership%20WP.doc

[2] QoSForum, “The need for QoS ” , 1999; http://www.qosforum.com/tech_resources.htm

[3] QoSForum, “The IP FAQ QoS Forum White Paper ” , 1999; http://www.qosforum.com/docs/faq/faq.pdf

[4] N. Bhatti and R. Friedrich, “Web Server Support for Tiered Services ” , December 1999; http://www.hpl.hp.com/techreports/1999/HPL-1999-160.html

[5] Agilent, “Agilent Firehunter Service Models and Baselining ” , 1999; http://www.firehunter.com/library/measure2.htm

[6] D. Caswell and S. Ramanathan, “Using Service Models for Management of Internet Services ” , 1999; http://www.hpl.hp.com/techreports/1999/HPL-1999-43.html

[7] G. D. Rodosek and T. Kaiser, “Determining the Availability of Distributed Applications ” , vol. Integrated Network Management V: Chapman & Hall, 1997;

[8] S. Adler, “The Slashdot Effect ” , 1998; http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html

[9] N. Kausar, B. Briscoe, and J. Crowcroft, “A charging model for Sessions on the Internet,” presented at Fourth IEEE Symposium on Computers and Communications, Sharm El Sheikh, Egypt, 1998.

[10] J. W. Hong, J. S. Kim, and J. T. Park, “A CORBA-based Quality-of-service Management Framework for Distributed Multimedia Services and Applications ” in *Proc. of IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1998;

[11] F. Siqueira and V. Cahill, “Quartz: Supporting QoS-Constrained Services in Heterogeneous Environments ” . Trinity College Dublin, January 1999; ftp://ftp.cs.tcd.ie/pub/tech-reports/reports.99/TCD-CS-1999-01.pdf

[12] H. Packard, “Service Level Agreements - An Emerging Trend in the Internet Services Market ” , 1998; http://www.firehunter.com/library/agreement.htm

[13] R. Friedrich, “Providing QoS for E-Business Providing QoS for E-Business Applications ” , October 1998; http://www.hpl.hp.com/org/isal/research/slides/ispcon98.PDF

[14] FORM, “Engineering a Co-operative Inter-Enterprise Management Framework Supporting Dynamic Federated Organisations Management ” : The FORM Consortium, 2000;

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1 ” : The Internet Society, June 1999; http://www.ietf.org/rfc/rfc2616.txt

[16] P. Flynn, “Frequently Asked Questions about the Extensible Markup Language ” ; http://www.ucc.ie/xml/

[17] JavaSoft, “Java XML Tutorial ” , 2000; http://java.sun.com/xml/tutorial_intro.html

[18] M. Asawa, “Measuring and Analyzing Service Levels: A Scalable Passive Approach ” , 1998; http://www.hpl.hp.com/techreports/97/HPL-97-138.html

[19] “Wilbur - HTML 3.2 ” : Web Design Group, 1997; http://www.htmlhelp.com/reference/wilbur/block/form.html

[20] W3C, “Jigsaw Frequently Asked Questions ” , 2000; http://www.w3.org/Jigsaw/Doc/FAQ.html

[21] D. Wessels and K. Claffey, “ICP and the Squid Web Cache ” in *IEEE Journal on Selected Areas In Communication*, vol. 16: IEEE, 1998, pp. 345-357;

[22] W3C, “Querying and Transforming XML,” , 1998.

8 APPENDICES

8.1 APPENDIX A

8.1.1 SLA DTD

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT SLA (GENERAL-PARAMETERS, QOS-PARAMETERS)>

<!ELEMENT GENERAL-PARAMETERS (SERVICE-ID, CONTRACT-ID, SERVICE-NAME, SERVICE-
AVAILABILITY, SERVICE-PERFORMANCE, SLA-VIOLATION-ACTION, SLA-REPORTING-INFO,
SERVICE-REQUESTS-PER-SECOND-LIMIT, SERVICE-ERROR-HEADER)>

<!ELEMENT SERVICE-ID EMPTY>
<!ATTLIST SERVICE-ID
        VALUE CDATA #REQUIRED
>

<!ELEMENT CONTRACT-ID EMPTY>
<!ATTLIST CONTRACT-ID
        VALUE CDATA #REQUIRED
>

<!ELEMENT SERVICE-NAME EMPTY>
<!ATTLIST SERVICE-NAME
        VALUE CDATA #REQUIRED
>

<!ELEMENT SERVICE-AVAILABILITY EMPTY>
<!ATTLIST SERVICE-AVAILABILITY
        REQUIRED-AVAILABILITY-PERCENTAGE CDATA #REQUIRED
```

```
>

<!ELEMENT SERVICE-PERFORMANCE EMPTY>
<!ATTLIST SERVICE-PERFORMANCE
        REQUIRED-COMPLIANCE-PERCENTAGE CDATA #REQUIRED
>

<!ELEMENT SLA-VIOLATION-ACTION (NOTIFY-SLA-CENTER, NOTIFY-ASP)>

<!ELEMENT NOTIFY-SLA-CENTER EMPTY>
<!ATTLIST NOTIFY-SLA-CENTER
        ACTION CDATA # IMPLIED
        URL CDATA #IMPLIED
>

<!ELEMENT NOTIFY-ASP EMPTY>
<!ATTLIST NOTIFY-ASP
        ACTION CDATA #IMPLIED
        URL CDATA #IMPLIED
>

<!ELEMENT SLA-REPORTING-INFO (REPORT-PERIOD, REPORT-FREQUENCY)>

<!ELEMENT REPORT-PERIOD EMPTY>
<!ATTLIST REPORT-PERIOD
        VALUE CDATA #REQUIRED
>

<!ELEMENT REPORT-FREQUENCY EMPTY>
<!ATTLIST REPORT-FREQUENCY
        VALUE CDATA #REQUIRED
>

<!ELEMENT SERVICE-REQUESTS-PER-SECOND-LIMIT EMPTY>
<!ATTLIST SERVICE-REQUESTS-PER-SECOND-LIMIT
        VALUE CDATA #REQUIRED
```

```
>

<!ELEMENT SERVICE-ERROR-HEADER EMPTY>
<!ATTLIST SERVICE-ERROR-HEADER
        VALUE CDATA #REQUIRED
>

<!ELEMENT QOS-PARAMETERS (REFERED-OBJECTS-PERFORMANCE-PARAMETERS, SERVICE-
MAPPINGS)>

<!ELEMENT REFERED-OBJECTS-PERFORMANCE-PARAMETERS (REFERED-OBJECTS-PERFORMANCE-
PARAMETER+)>


<!ELEMENT REFERED-OBJECTS-PERFORMANCE-PARAMETER EMPTY>
<!ATTLIST REFERED-OBJECTS-PERFORMANCE-PARAMETER
        NAME CDATA #REQUIRED
        VALUE CDATA #REQUIRED
        FUNCTION (MAX | MIN) #REQUIRED
>

<!ELEMENT SERVICE-MAPPINGS (URL-PARAMETERS-MAPPING+)>

<!ELEMENT URL-PARAMETERS-MAPPING (REQUEST, PERFORMANCE-PARAMETERS, MONITOR-
REFERED-OBJECTS)>
<!ATTLIST URL-PARAMETERS-MAPPING
        MAPPING-NAME CDATA #REQUIRED
        MAPPING-ID CDATA #REQUIRED
>

<!ELEMENT REQUEST EMPTY>
<!ATTLIST REQUEST
        METHOD (GET | HEAD | POST | PUT | DELETE | TRACE | CONNECT) #REQUIRED
        URL CDATA #REQUIRED
>
```

```
<!ELEMENT PERFORMANCE-PARAMETERS (PERFORMANCE-PARAMETER+)>

<!ELEMENT PERFORMANCE-PARAMETER EMPTY>
<!ATTLIST PERFORMANCE-PARAMETER
        NAME CDATA #REQUIRED
        VALUE CDATA #REQUIRED
        FUNCTION (MAX | MIN) #REQUIRED
>

<!ELEMENT MONITOR-REFERED-OBJECTS EMPTY>
<!ATTLIST MONITOR-REFERED-OBJECTS
        VALUE CDATA #REQUIRED
>
```

8.2 APPENDIX B

8.2.1 SERVICES DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This DTD is used to describe the services that the proxy should monitor -->
<!ELEMENT SERVICE-SLA-INFO (SERVICE-ID, CONTRACT-ID)>
<!ELEMENT SERVICES (SERVICE-SLA-INFO)>
<!-- SERVICES-REFRESH-PERIOD is the time in milliseconds that the SLA Agent rechecks services-->
<!ATTLIST SERVICES
        SERVICES-REFRESH-PERIOD CDATA #REQUIRED
>
<!-- SERVICE-ID is the service identifier -->
<!ELEMENT SERVICE-ID EMPTY>
<!ATTLIST SERVICE-ID
        VALUE CDATA #REQUIRED
>
<!-- CONTRACT-ID is the contract identifier -->
<!ELEMENT CONTRACT-ID EMPTY>
<!ATTLIST CONTRACT-ID
        VALUE CDATA #REQUIRED
>
```

8.3 APPENDIX C

8.3.1 SLA for Yahoo Mail Case Study

```
<?xml version="1.0" encoding="UTF-8"?>
<SLA>
	<GENERAL-PARAMETERS>
		<SERVICE-ID VALUE="100"/>
		<CONTRACT-ID VALUE="1"/>
		<SERVICE-NAME VALUE="Yahoo Mail Web based Email Service: Case Study"/>
		<SLA-VIOLATION-ACTION>
			<NOTIFY-SLA-CENTER ACTION="NO"/>
			<NOTIFY-ASP ACTION="NO"/>
		</SLA-VIOLATION-ACTION>
		<SERVICE-AVAILABILITY REQUIRED-AVAILABILITY-PERCENTAGE="98"/>
		<SERVICE-PERFORMANCE REQUIRED-COMPLIANCE-PERCENTAGE="95"/>
		<SERVICE-REQUESTS-PER-SECOND-LIMIT VALUE="10"/>
		<SERVICE-ERROR-HEADER VALUE="YAHOO-MAIL-SERVICE-ERROR"/>
		<SLA-REPORTING-INFO>
			<REPORT-PERIOD VALUE="3000000000"/>
			<REPORT-FREQUENCY VALUE="900000"/>
		</SLA-REPORTING-INFO>
	</GENERAL-PARAMETERS>
	<QOS-PARAMETERS>
		<REFERED-OBJECTS-PERFORMANCE-PARAMETERS>
			<REFERED-OBJECTS-PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="2000" FUNCTION="MAX"/>
			<REFERED-OBJECTS-PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="1" FUNCTION="MIN"/>
		</REFERED-OBJECTS-PERFORMANCE-PARAMETERS>
		<SERVICE-MAPPINGS>
		<URL-PARAMETERS-MAPPING MAPPING-NAME="Login to Mail" MAPPING-ID="1">
				<REQUEST METHOD="POST" URL="http://login.yahoo.com/config/login?"/>
				<PERFORMANCE-PARAMETERS>
					<PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="2000" FUNCTION="MAX"/>
					<PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="2" FUNCTION="MIN"/>
				</PERFORMANCE-PARAMETERS>
				<MONITOR-REFERED-OBJECTS VALUE="NO"/>
			</URL-PARAMETERS-MAPPING>

			<URL-PARAMETERS-MAPPING MAPPING-NAME="Read Mail" MAPPING-ID="2">
				<REQUEST METHOD="GET" URL="http://us.f1.mail.yahoo.com/ym/ShowLetter?"/>
```

```
                        <PERFORMANCE-PARAMETERS>
                                <PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="2000" FUNCTION="MAX"/>
                                <PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="2" FUNCTION="MIN"/>
                        </PERFORMANCE-PARAMETERS>
                        <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                </URL-PARAMETERS-MAPPING>

                <URL-PARAMETERS-MAPPING MAPPING-NAME="Compose Mail" MAPPING-ID="3">
                        <REQUEST METHOD="GET" URL="http://us.f1.mail.yahoo.com/ym/Compose?"/>
                        <PERFORMANCE-PARAMETERS>
                                <PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="5000" FUNCTION="MAX"/>
                                <PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="1" FUNCTION="MIN"/>
                        </PERFORMANCE-PARAMETERS>
                        <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                </URL-PARAMETERS-MAPPING>

                <URL-PARAMETERS-MAPPING MAPPING-NAME="Delete Mail" MAPPING-ID="4">
                        <REQUEST METHOD="POST" URL="http://us.f1.mail.yahoo.com/ym/ShowFolder?"/>
                        <PERFORMANCE-PARAMETERS>
                                <PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="5000" FUNCTION="MAX"/>
                                <PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="1" FUNCTION="MIN"/>
                        </PERFORMANCE-PARAMETERS>
                        <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                </URL-PARAMETERS-MAPPING>


                <URL-PARAMETERS-MAPPING MAPPING-NAME="Send Mail" MAPPING-ID="5">
                        <REQUEST METHOD="POST" URL="http://us.f1.mail.yahoo.com/ym/Compose"/>
                        <PERFORMANCE-PARAMETERS>
                                <PERFORMANCE-PARAMETER NAME="ENTITY-UPLOAD-RATE"
VALUE="1" FUNCTION="MIN"/>
                        </PERFORMANCE-PARAMETERS>
                        <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                </URL-PARAMETERS-MAPPING>

                <URL-PARAMETERS-MAPPING MAPPING-NAME="Search Mail" MAPPING-ID="6">
                        <REQUEST METHOD="GET" URL="http://us.f1.mail.yahoo.com/ym/Search"/>
                        <PERFORMANCE-PARAMETERS>
                                <PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="5000" FUNCTION="MAX"/>
```

```
                                        <PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="1" FUNCTION="MIN"/>
                                </PERFORMANCE-PARAMETERS>
                                <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                        </URL-PARAMETERS-MAPPING>

                        <URL-PARAMETERS-MAPPING MAPPING-NAME="Search Mail Results" MAPPING-ID="7">
                                <REQUEST METHOD="POST" URL="http://us.f1.mail.yahoo.com/ym/Search"/>
                                <PERFORMANCE-PARAMETERS>
                                        <PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="5000" FUNCTION="MAX"/>
                                        <PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="2" FUNCTION="MIN"/>
                                </PERFORMANCE-PARAMETERS>
                                <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                        </URL-PARAMETERS-MAPPING>

                        <URL-PARAMETERS-MAPPING MAPPING-NAME="Logout of Mail" MAPPING-ID="8">
                                <REQUEST METHOD="GET" URL="http://us.f1.mail.yahoo.com/ym/ym/Logout?"/>
                                <PERFORMANCE-PARAMETERS>
                                        <PERFORMANCE-PARAMETER NAME="HEADER-RESPONSE-TIME"
VALUE="5000" FUNCTION="MAX"/>
                                        <PERFORMANCE-PARAMETER NAME="ENTITY-DOWNLOAD-RATE"
VALUE="2" FUNCTION="MIN"/>
                                </PERFORMANCE-PARAMETERS>
                                <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                        </URL-PARAMETERS-MAPPING>


                        <URL-PARAMETERS-MAPPING MAPPING-NAME="Attach to Maill" MAPPING-ID="9">
                                <REQUEST METHOD="GET" URL="http://us.f1.mail.yahoo.com/ym/Attachments?"/>
                                <PERFORMANCE-PARAMETERS>
                                        <PERFORMANCE-PARAMETER NAME="ENTITY-UPLOAD-RATE"
VALUE="1" FUNCTION="MIN"/>
                                </PERFORMANCE-PARAMETERS>
                                <MONITOR-REFERED-OBJECTS VALUE="NO"/>
                        </URL-PARAMETERS-MAPPING>
                </SERVICE-MAPPINGS>
        </QOS-PARAMETERS>
</SLA>
```

8.4 Appendix D

8.4.1 Service Descriptor for Yahoo Mail Case Study

```
<?xml version="1.0" encoding="UTF-8"?>
<SERVICES SERVICES-REFRESH-PERIOD="300000">
    <SERVICE-SLA-INFO>
        <SERVICE-ID VALUE="100"/>
        <CONTRACT-ID VALUE="1"/>
    </SERVICE-SLA-INFO>
</SERVICES>
```

8.5 Appendix E

8.5.1 SLA Report for Yahoo Mail Case Study:

SLA Report

Report Details

Date of Report: Sat Sep 9 22:00:13 GMT 2000

Service Usage From: Sat Sep 9 21:00:13 GMT 2000 to Sat Sep 9 22:00:13 GMT 2000

Service Name: Yahoo Mail Web based Email Service: Case Study

Service ID: 100

Contract ID: 1

Service Compliance Summary

- Performance Complies as Specified in SLA? NO [Breakdown]
- Availability Complies as Specified in SLA? NO [Breakdown]

Service Usage

Total Number of Requests to Service: 31

Service Usage Breakdown By Mappings:

- Mapping Name: Login to Mail
  - Number of Requests: 2
  - Percentage of Total Requests to Service: 6%
- Mapping Name: Read Mail
  - Number of Requests: 3
  - Percentage of Total Requests to Service: 9%
- Mapping Name: Compose Mail
  - Number of Requests: 7
  - Percentage of Total Requests to Service: 22%
- Mapping Name: Delete Mail
  - Number of Requests: 8
  - Percentage of Total Requests to Service: 25%
- Mapping Name: Send Mail
  - Number of Requests: 3
  - Percentage of Total Requests to Service: 9%
- Mapping Name: Search Mail
  - Number of Requests: 2
  - Percentage of Total Requests to Service: 6%
- Mapping Name: Search Mail Results
  - Number of Requests: 1
  - Percentage of Total Requests to Service: 3%
- Mapping Name: Logout of Mail
  - Number of Requests: 0
  - Percentage of Total Requests to Service: 0%
- Mapping Name: Attach to Maill
  - Number of Requests: 5
  - Percentage of Total Requests to Service: 16%

Service Performance

Required Percentage Performance Compliance Specified in SLA: 95%

Performance Compliance Percentage in Period: 80%

- Service Performance Breakdown By Mapping

- Mapping Name: Login to Mail
  - Parameter Name: HEADER-RESPONSE-TIME [Definition]
    - Required Value of <2000
    - Percentage of Requests Compliant: 100%
    - Parameter Thresholds Observed
      - Minimum: 1211
      - Average: 1236
      - Maximum: 1261
  - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
    - Required Value of >2
    - Percentage of Requests Compliant: 100%
    - Parameter Thresholds Observed
      - Minimum: 12
      - Average: 12
      - Maximum: 12
- Mapping Name: Read Mail
  - Parameter Name: HEADER-RESPONSE-TIME [Definition]
    - Required Value of <2000
    - Percentage of Requests Compliant: 33%
    - Parameter Thresholds Observed
      - Minimum: 1871
      - Average: 2274
      - Maximum: 2751
  - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
    - Required Value of >2

        - Percentage of Requests Compliant: 33%
        - Parameter Thresholds Observed
            - Minimum: 1
            - Average: 1
            - Maximum: 2
- Mapping Name: Compose Mail
    - Parameter Name: HEADER-RESPONSE-TIME [Definition]
        - Required Value of <5000
        - Percentage of Requests Compliant: 83%
        - Parameter Thresholds Observed
            - Minimum: 2201
            - Average: 3031
            - Maximum: 6811
    - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
        - Required Value of >1
        - Percentage of Requests Compliant: 100%
        - Parameter Thresholds Observed
            - Minimum: 1
            - Average: 1
            - Maximum: 2
- Mapping Name: Delete Mail
    - Parameter Name: HEADER-RESPONSE-TIME [Definition]
        - Required Value of <5000
        - Percentage of Requests Compliant: 100%
        - Parameter Thresholds Observed
            - Minimum: 2041
            - Average: 2152
            - Maximum: 2311
    - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
        - Required Value of >1
        - Percentage of Requests Compliant: 75%

        - Parameter Thresholds Observed
            - Minimum: 0
            - Average: 1
            - Maximum: 2
- Mapping Name: Send Mail
    - Parameter Name: ENTITY-UPLOAD-RATE [Definition]
        - Required Value of >1
        - Percentage of Requests Compliant: 100%
        - Parameter Thresholds Observed
            - Minimum: 2
            - Average: 3
            - Maximum: 3
- Mapping Name: Search Mail
    - Parameter Name: HEADER-RESPONSE-TIME [Definition]
        - Required Value of <5000
        - Percentage of Requests Compliant: 100%
        - Parameter Thresholds Observed
            - Minimum: 2031
            - Average: 2086
            - Maximum: 2141
    - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
        - Required Value of >1
        - Percentage of Requests Compliant: 100%
        - Parameter Thresholds Observed
            - Minimum: 1
            - Average: 1
            - Maximum: 2
- Mapping Name: Search Mail Results
    - Parameter Name: HEADER-RESPONSE-TIME [Definition]
        - Required Value of <5000
        - Percentage of Requests Compliant: 0%

        - Parameter Thresholds Observed
            - Minimum: 6871
            - Average: 6871
            - Maximum: 6871
    - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
        - Required Value of >2
        - Percentage of Requests Compliant: 0%
        - Parameter Thresholds Observed
            - Minimum: 1
            - Average: 1
            - Maximum: 1
- Mapping Name: Logout of Mail
    - Parameter Name: HEADER-RESPONSE-TIME [Definition]
        - Required Value of <5000
        - Percentage of Requests Compliant: 0%
        - Parameter Thresholds Observed
            - Minimum: 0
            - Average: 0
            - Maximum: 0
    - Parameter Name: ENTITY-DOWNLOAD-RATE [Definition]
        - Required Value of >2
        - Percentage of Requests Compliant: 0%
        - Parameter Thresholds Observed
            - Minimum: 0
            - Average: 0
            - Maximum: 0
- Mapping Name: Attach to Maill
    - Parameter Name: ENTITY-UPLOAD-RATE [Definition]
        - Required Value of >1
        - Percentage of Requests Compliant: 0%
        - Parameter Thresholds Observed

- Minimum: 0
- Average: 0
- Maximum: 0

Service Availability

Required Percentage Availability Compliance Specified in SLA: 98%

Total Number of Failed Requests to Service: 2

Percentage of Requests that Succeded: 94%

- Breakdown of Failure Type
    - Server Error: 0
    - Service Error: 0
    - HTTPException: 2

Service Availability Breakdown By Mappings:

- Mapping Name : Login to Mail
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0
- Mapping Name : Read Mail
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0
- Mapping Name : Compose Mail
    - Number of Failed Requests: 1
    - Percentge of Total Failed Requests: 50%
    - Breakdown By Failure Type

        - Server Error: 0
        - Service Error: 0
        - HTTPException: 1
- Mapping Name : Delete Mail
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0
- Mapping Name : Send Mail
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0
- Mapping Name : Search Mail
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0
- Mapping Name : Search Mail Results
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0

- Mapping Name : Logout of Mail
    - Number of Failed Requests: 0
    - Percentge of Total Failed Requests: 0%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 0
- Mapping Name : Attach to Maill
    - Number of Failed Requests: 1
    - Percentge of Total Failed Requests: 50%
    - Breakdown By Failure Type
        - Server Error: 0
        - Service Error: 0
        - HTTPException: 1

Parameter Definitions

- Parameters Name: HEADER-RESPONSE-TIME
    - Definition: Response time in milliseconds to receive a reply header from time that request is sent
- Parameters Name: ENTITY-DOWNLOAD-RATE
    - Definition: Transfer rate in Kbps of reply body from service
- Parameters Name: ENTITY-UPLOAD-RATE
    - Definition: Transfer rate in Kbps of request body to service

8.6 Appendix F

8.6.1 Table of Parameters Logged HTTPQoS Filter

Parameter Name	Description
RequestID	The request ID
requestURL	The URL of the request
requestReferer	The Referrer URL of the request
requestRefererID	The request ID of the Referrer URL
requestTime	The time of the reply is being sent to the server
requestHasOutputStream	Indicates if the request has an output stream
requestContentLength	The content length of the request body
requestContentType	The content type of the request
requestMethod	The request Method
requestBodyStartTime	The start time of reading of the request output stream
requestBodyEndTime	The end time of reading the request output stream
requestException	Any exception that the request raises
replyTime	The time of the reply is received from the server
replyStatus	The HTTP status code of the reply
replyHasInputStream	Indicates if the request has an input stream
replyContentLength	The content length of the reply body
replyContentType	The Content type of the reply
replyBodyStartTime	The start time reading of the reply input stream
replyBodyEndTime	The end time reading of the reply input stream
userAgent	The user-agent of the Web browser making the request
serviceID	The service identifier that the request was logged for
contractID	The contract identifier that the request was logged for
mappingID	The mapping identifier that the request was logged for
serviceErrorHeader	The value of the service error header returned by ASP