A Throttling Layer-7 Web Switch

Final Report

Written by:
James Furness

Supervised by:
Dr. Julie McCann

Second Marker:
Dr. Dan Chalmers

Project Homepage:
http://www.base6.com/mercury

Abstract

The world-wide web has grown rapidly from a research tool in 1991 to a part of everyday life for most people. In this same time the content available has evolved from bland text pages with few images to rich multimedia, interactive and often dynamically generated pages. An ever-growing strain is placed on popular websites to handle increasing volumes of traffic and deliver pages quickly to users with a constantly decreasing attention span.

The primary aim of this project is to produce a web switch, which enables a pool of web servers to present themselves as a single virtual server. This web switch supports throttling or downgrading of content under high load situations, attempting to ensure that "everybody gets something" rather than "some people get everything and some people get nothing".

Acknowledgements

I would like to thank my supervisor Julie McCann, my second marker Dan Chalmers and Gawesh Jawaheer for their supervision and invaluable assistance during the course of this project, providing encouragement, ideas and reference sources.

I would also like to thank Tim Southerwood, Matt Johnson and Duncan White for their willingness to discuss my ideas and provide guidance on kernel modification, testing and other technical aspects of this project.

Contents

1 Introduction 7
1.1 Motivation 7
1.2 Project Goals 8
1.3 Report Structure 8
2 Background 11
2.1 Introduction 11
2.2 HTTP: The world-wide web 11
2.2.1 Protocol Overview 11
2.2.2 HTTP 1.1 12
2.2.3 CGI Scripts 13
2.3 Effects of High Demand 13
2.3.1 Overall Demand 13
2.3.2 Flash Crowd Events and DoS Attacks 14
2.4 Coping with High Demand 14
2.4.1 Slowing Requests 15
2.4.2 Increasing Capacity 15
2.4.3 Increasing Processing Speed 23
2.5 Summary 23
3 Design Overview 25
3.1 Introduction 25
3.2 Terminology 25
3.3 System Architecture 25
3.4 Routing Layer 26
3.5 Control Layer 26
3.5.1 Control Layer Interface 27
3.5.2 Policy Engine 28
3.5.3 Virtual File System 29
3.5.4 Configuration Module 29
3.5.5 Summary 30
3.6 Initial Monitoring data and metadata 30
3.6.1 Monitoring data 31
3.6.2 Metadata 34
3.7 Standardising Response Times 35
3.7.1 Introduction 35
3.7.2 Implementation 35
3.8 Summary 37
4 Detailed Design 39
4.1 Introduction 39
4.2 Implementation Tools and Techniques 39
4.3 Control Layer 40
4.3.1 XML Configuration Format 40
4.3.2 Control Layer Interface (mercury.urimapper) 41
4.3.3 Policy Engine (mercury.logic) 43
4.3.4 Policy Engine System Monitors (mercury.monitors) 45
4.3.5 Virtual File System (mercury.vfs) 45
4.3.6 Configuration Module (mercury.config) 53
4.3.7 Debugging and logging (mercury.debug) 57
4.3.8 Initial Dispatching Algorithm 57
4.4 Routing Layer 60
4.4.1 TCP Hand-off 60
4.4.2 TCP Gateway 60

4.4.3 Evaluation of Proxy Servers 61
4.5 Portability 64
4.6 Scalability and Resilience 64
4.7 Summary 66

5 Testing 67
5.1 Introduction 67
5.2 Unit testing 67
5.3 Integration testing 67
5.4 Effectiveness testing 67
5.4.1 Test setup 68
5.4.2 List of tests conducted 70

6 Evaluation 85
6.1 Introduction 85
6.2 Summary of Goals 85

7 Conclusion 87
7.1.1 Limitations 87
7.1.2 Extensions 88
7.1.3 Summary of achievements 88

8 Bibliography 91

1 Introduction

1.1 Motivation

"The overall increase in traffic on the World Wide Web is augmenting user-perceived response times from popular Web sites […] System platforms that do not replicate information content cannot provide the needed scalability to handle large traffic volumes and to match rapid and dramatic changes in the number of clients." [1]

Since its birth in 1990, the world-wide web has shown phenomenal growth due to its ideal suitability as a mechanism for the rapid dissemination of information. The mass availability of information through the world-wide web has spearheaded the growth of Internet access and this has in turn further encouraged the growth of the world-wide web.

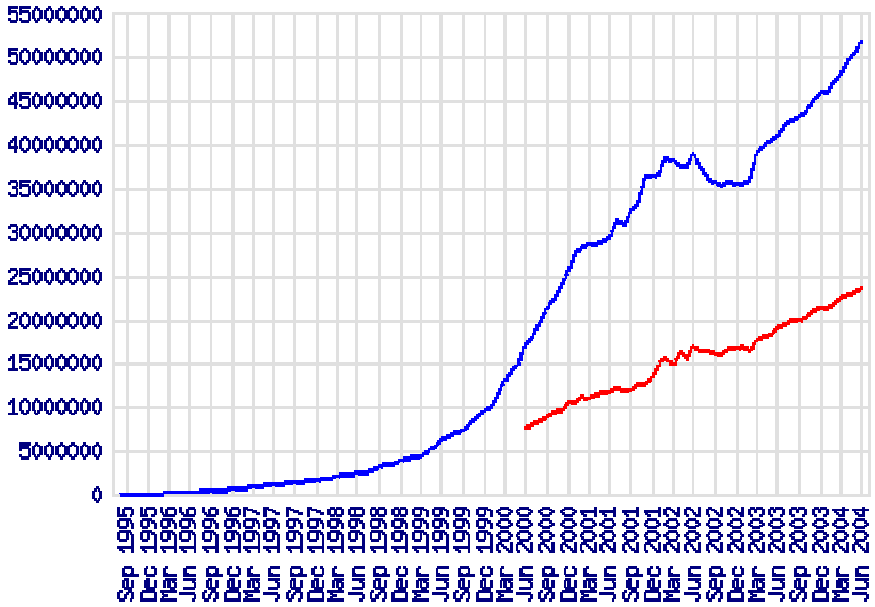

Figure 1 Growth of the World-Wide Web Aug 1995-Jun 2004 (Upper line indicates hostnames, lower indicates unique hosts) [2]

In addition, Internet connections used by end users have increased dramatically in speed from 9600bps modems that were the state of the art in 1990 to the broadband services of today offering speeds of 2Mbps and above. Backbone connections between Internet providers are similarly increasing in speed. Users

are able to browse the world-wide web with increased data rates and reduced latency.

These factors have placed a much greater importance on the ability of a popular web site to handle large numbers of concurrent users whilst keeping user-perceived response times within a limit acceptable to the user. This limit is continually decreasing as Internet connection speeds increase.

Optimising web sites to handle these demands has been a target of research since 1994 and many of the problems involved have been addressed, however a great deal of research is still ongoing.

1.2 Project Goals

This project attempts to create a system facilitating a pool of web servers to present themselves as a single virtual server with the following features:

- **Throttling (Primary Goal)**: The system should attempt to maximise availability and response times with the resources available to it by downgrading pages to alternative versions requiring less resources when under high demand. (E.g. text only pages instead of multimedia pages when a large number of concurrent users are using the site)
- **Load balancing**: The system should balance the load of connections between the servers in the pool, allowing scalability under load by distributing requests amongst the pool.
- **Heterogeneous pool**: The system should allow the pool to be heterogeneous, where a particular resource may be on some but not all of the pool servers. It should also allow servers to join and leave the pool at any time.
- **Adaptable**: The system should be as flexible as possible and provide a framework to allow a variety of load balancing/throttling algorithms to be used

1.3 Report Structure

Chapter 2 provides an overview of the technologies involved and the current state of the art
Chapter 3 contains a high-level overview of the chosen design
Chapter 4 contains a detailed view of the chosen design

Chapter 5 details the experimental testbed, data collection procedures and testing carried out on the system

Chapter 6 evaluates the performance of the system

Chapter 7 summarises the achievements of the project, identifies limitations and possible further work.

2 Background

2.1 Introduction

This chapter presents an overview of the key technologies, concepts and existing research related to the project:

- Firstly an overview of HTTP is given, the underlying protocol driving the world-wide web.
- Secondly the problems caused by high demand for a web site are discussed.
- Finally an overview is presented of currently available techniques to overcome the problems of high demand.

2.2 HTTP: The world-wide web

This section presents a brief overview of HTTP, the underlying protocol that drives the world-wide web. Additionally some more advanced points relevant to this project have been included.

HTTP stands for HyperText Transfer Protocol and is the network protocol used to deliver files and data over the world-wide web. The first version was developed in 1990 at CERN by Tim Berners-Lee.

2.2.1 Protocol Overview

The standard method of addressing files is to use a Uniform Resource Locator (URL) to identify a location on the server. This is a type of Uniform Resource Indicator (URI). URIs are typically of the form *`service:parameters`*. URLs are typically of the form *`http://host:port/path/file.html`*. Often the port is omitted and defaults to the standard HTTP port, 80.

HTTP generally communicates over a TCP/IP socket connection and is connectionless and stateless. It is based upon a request/response paradigm, and in its most basic form consists of the following steps:

1. Client establishes a TCP connection to the server host and port given in the URL
2. Send the HTTP Request to the server

3. Receive the HTTP Response
4. Close the TCP connection

The HTTP Request consists of a request line specifying the operation (Most commonly GET, HEAD or POST), requested path and protocol version. This is followed by zero or more request headers specifying additional information and then a blank line. In the case of a POST request the headers are followed data. A typical request might look like this:

```
GET /test.txt HTTP/1.1
Host: www.doc.ic.ac.uk
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)
```

The HTTP Response is structured similarly, with the first line specifying the protocol version, a numeric status code and description. This is followed by response headers, a blank line and then the content of the response. A typical response might look like this:

```
HTTP/1.1 200 OK
Date: Fri, 20 Feb 2004 13:31:00 GMT
Server: Apache/1.2.0
Content-Type: text/plain

This is a test document.
```

Note that the numeric status code is machine-readable and the first digit corresponds to the category of response (For example 2xx indicates a success)

2.2.2 HTTP 1.1

The initial version, HTTP 0.9 only supported raw data transfer, and rapidly became a de-facto standard on the Internet. The first official version, HTTP 1.0 was defined by RFC 1945 in 1996 and added content type negotiation.

Several major problems existed in this version and in 1999 HTTP 1.1 was defined by RFC 2616. Improvements include:

- **Persistent connections**: Most HTML pages reference other objects such as images; under HTTP 1.0 a new connection is created for each object so a page with N referenced objects requires N+1 connections. Setting up a

new TCP/IP connection causes an unnecessary overhead, HTTP 1.1 uses persistent connections which allow several requests to be sent over one connection

- **Hostname identification**: A Host: header is added to all requests allowing one IP address to be allocated to multiple domain names
- **Proxy support**: HTTP 1.1 adds additional headers to help proxies determine how long to keep documents in their cache
- **Byte ranges**: The client can specify a byte range to be retrieved instead of a whole document
- **Compression**: Compression of documents can be negotiated between client and server
- **Pipelining**: Several requests can be sent on a persistent connection without waiting for responses. The responses can then be sent together, maximising packet sizes and increasing network efficiency.

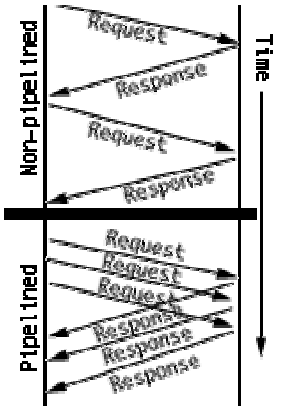

Figure 2 Pipelining [3]

2.2.3 CGI Scripts

A CGI script is a program that runs on the web server and generates a dynamic response to the client's request. It interfaces with the web server via the Common Gateway Interface (CGI) standard [11]

2.3 Effects of High Demand

2.3.1 Overall Demand

A popular website has a high average request rate since it is being accessed simultaneously by many people. The web site must be able to endure a higher request rate than the average in order to cope with peaks in demand. For a typical website these peaks can be significantly higher than the average, since

the majority of users for a site are often in the same time zone, and the request rate is low during the night and high during the day.

2.3.2 Flash Crowd Events and DoS Attacks

These events cause a significant load to be suddenly and unexpectedly placed on a web site.

A flash crowd event is caused by a huge number of users trying to load a website simultaneously (For example the September 11th terrorist attacks which caused several major news sites to be unavailable). A Denial of Service (DoS) attack is caused by a user maliciously sending a large volume of requests to a website in order to disrupt its service.

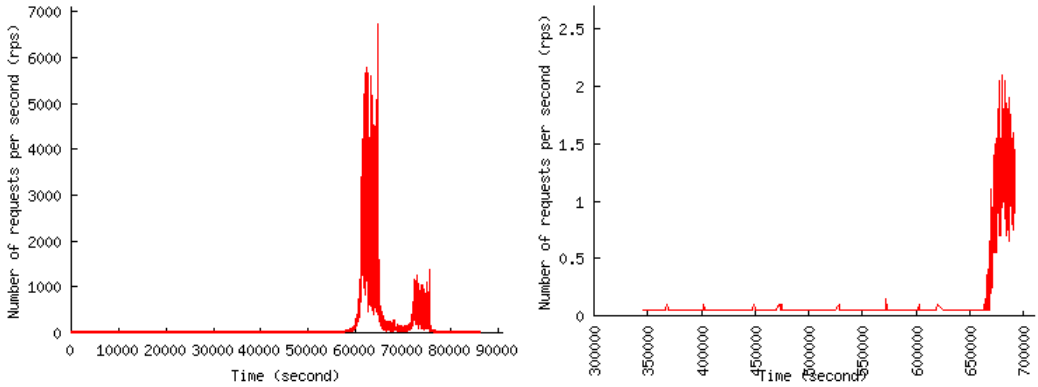

Figure 3 Traffic volume for a Flash Event (left) and a DoS attack (right) [20]

Both flash crowd events and DoS attacks cause degradation to the service, or complete failure of the website.

2.4 Coping with High Demand

In order to better cope with the types of High Demand described above, a web site essentially has three options:

- Slow the incoming request rate down
- Increase capacity in order to cope with the peak demand
- Increase the speed at which requests are processed

2.4.1 Slowing Requests

In [22], the network early warning system (NEWS) is proposed. This relies upon detecting the flash crowd event and then employs an adaptive rate limiting technique to reduce the request rate to an acceptable value.

2.4.2 Increasing Capacity

This section describes currently available technologies to increase the capacity of a web site in order to cope with higher demand. This section consists mainly of a summary of the paper *"The State of the Art in Locally Distributed Web-Server Systems"* [1] which presents a comprehensive and detailed overview of the technologies available.

An overview of scalable web-server systems

A popular web site faces a constant need to increase capacity. This requires the web system serving the site to be scalable. Web system scalability is defined as the ability to support large numbers of accesses and resources whilst still providing adequate performance.

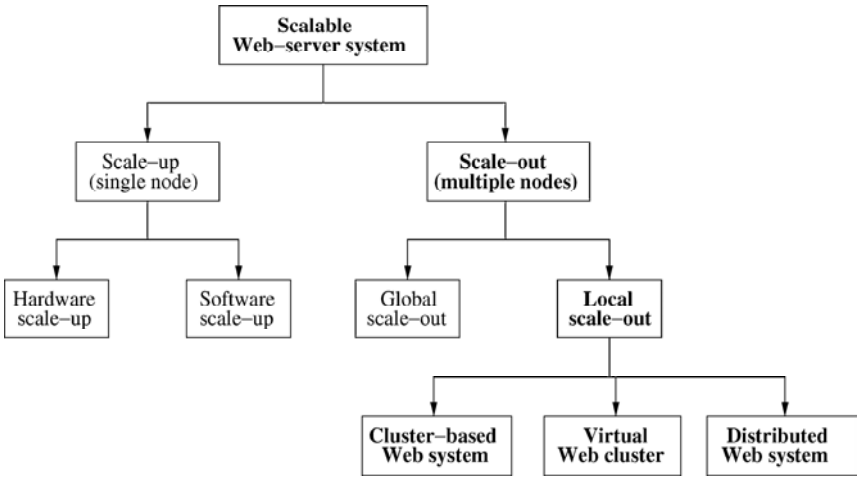

Figure 4 Architecture solutions for scalable Web-server systems [1]

To build scalable web content delivery architectures, two major options exist:

- ***Scale-up* (Single node)**: Continue to serve resources from one physical server and upgrade the server hardware and/or software to cope with higher traffic.
- ***Scale-out* (Multiple nodes)**: Switch from using one physical server to using multiple servers.

A *scale-up* solution is limited by the resources available on a single physical server, whereas a *scale-out* solution ideally allows more nodes to be "hot-plugged" into the solution as traffic increases. A number of reasonably powerful servers are also likely to be cheaper than one hugely powerful server.

Scale-out solutions are further grouped into two major categories: *local scale-out* where the servers are co-located at a single network location and *global scale-out* where the servers are geographically distributed.

This project is intended to present a cluster of co-located servers transparently as if they were a single virtual server. Hence *global scale-out* solutions will not be discussed further.

Local scale-out solutions can be further subdivided:

- ***Cluster-based web system***: A collection of servers that present a single system image to the outside (One DNS name and one virtual IP address, or *VIP*). Each node contains its own disk and a complete operating system. The front-end node, or *web switch* receives all inbound packets and routes them to some web-server node. This is also the only *local scale-out* solution that is capable of content-aware redirection.
- ***Virtual web cluster***: Similar to a cluster-based web system, but all nodes share the VIP such that each receives all inbound packets and filters them to decide whether to accept or discard them.
- ***Distributed web system***: A collection of servers that present multiple system images to the outside, the switching is facilitated either by the DNS server during the *lookup phase* (Where the DNS address is resolved to an IP address) or simply by explicitly instructing the client which IP address to use.

Again since this project is intended to present a cluster of servers transparently as one virtual server, and additionally to provide content-aware redirection (Due to the requirement for a heterogeneous server pool), only *cluster-based web systems* will be discussed further.

The state of the art in Cluster-based web systems

Cluster-based web systems consist of a collection of co-located servers interconnected through a single high-speed network that present a single system

image to the outside. Each server node of the cluster usually contains its own disk and a complete operating system.

The single system image is presented through one DNS name and one virtual IP address (VIP). This provides the sole interface from the cluster nodes to the Internet, and as such the architecture is completely transparent to the user and client application. The VIP address corresponds to the IP address of one front end node, or *web switch* (Multiple nodes sharing the same virtual IP address can also be used).

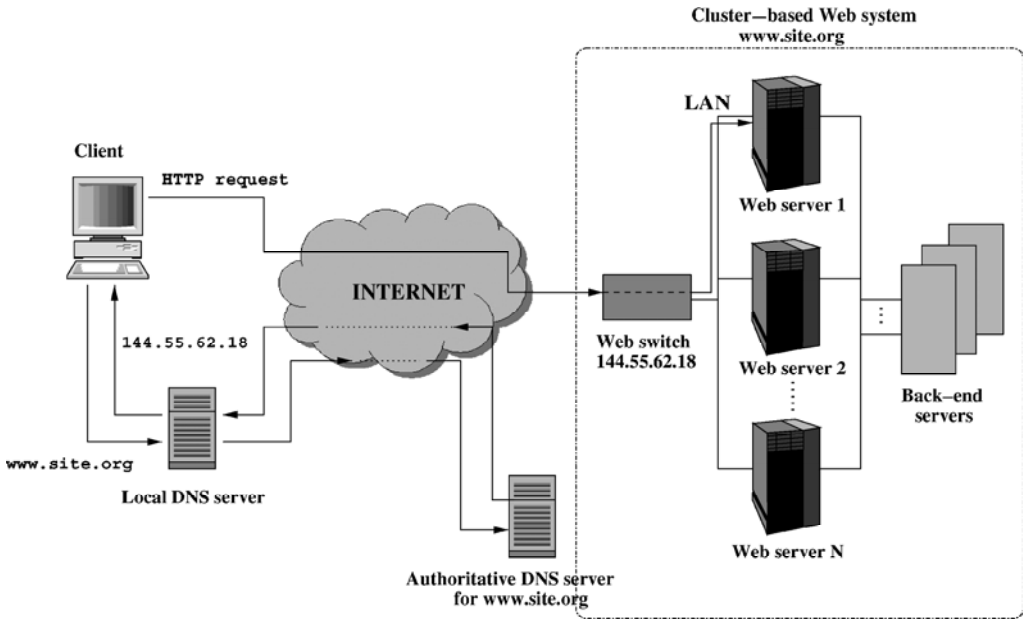

Figure 5 Architecture of a *cluster-based web system* [1]

The web switch is able to uniquely identify each node through a private address, either an IP address or a lower-layer MAC address. The key difference between *web switches* is the OSI protocol stack layer at which the web switch routes packets, at the transport layer (*layer-4*) or the application layer (*layer-7)*.

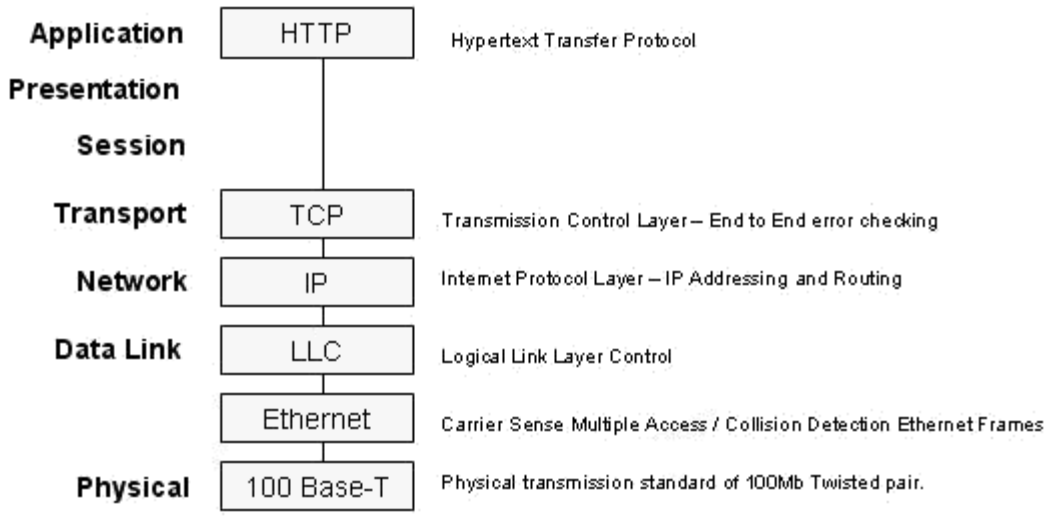

Figure 6 Example OSI Network Stack [4]

- ***Layer-4 web switches*** perform *content-blind routing* because they choose the target server during the establishment of the connection before the connection is actually opened. As such the redirection is efficient, but is unaware of the content of the client request.
- ***Layer-7 web switches*** perform *content-aware routing* since the switch establishes a complete TCP connection with the client and is able to examine the HTTP request. This is less efficient but provides more sophisticated dispatching.

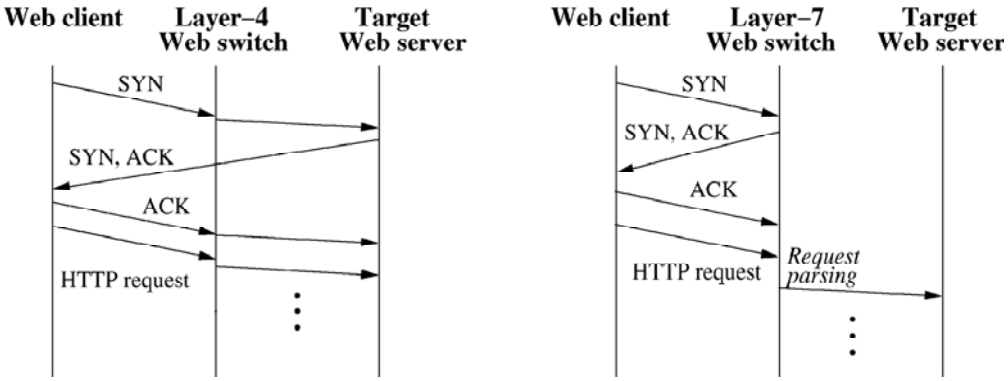

Figure 7 Operations of layer-4 routing (left) and layer-7 routing (right) [1]

Since this project requires content-aware redirection, only *layer-7 web switches* will be discussed further.

Layer-7 Web Switches

Layer-7 web switches work at the application layer. This requires the switch to establish a TCP connection with the client (I.e. the three-way handshake) and then receive the HTTP request at the application layer.

The various approaches to *layer-7* switching fall into two major groups: *one-way* architectures and *two-way* architectures.

In ***two-way* architectures**, outbound traffic must pass back through the *Web switch*. This approach has problems with scalability since the outbound bandwidth of the switch(s) is shared between all nodes in the cluster. Additionally note that the inbound traffic (the HTTP request) is likely to be many times smaller than the outbound traffic (the content of the response). This means that the rate at which responses can be sent is limited by the bandwidth through the web switch.

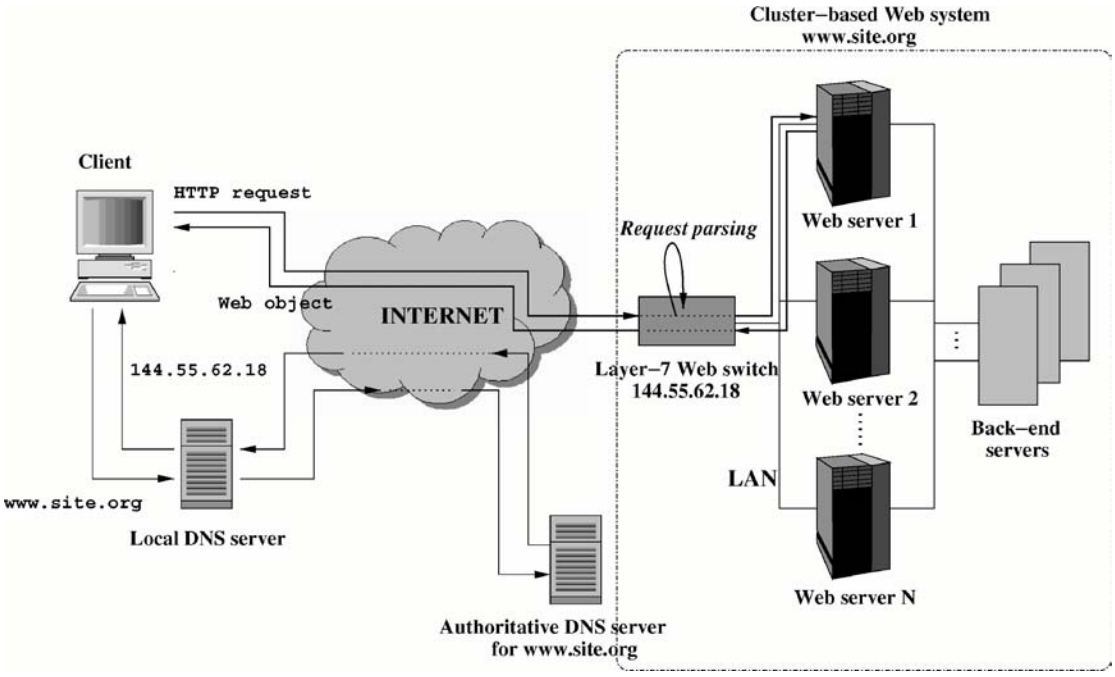

Figure 8 Layer-7 *two-way* architecture [1]

Two-way architectures include:

- **TCP Gateway**: A proxy running on the web switch at application level receives requests. It maintains a persistent connection with each web server and forwards each client request through a persistent connection to the appropriate web server. It then receives the response through the connection and forwards it to the client

- **TCP Splicing**: The previous approach is computationally expensive since all packets must flow up to the application layer. This approach forwards packets at the network layer instead – once the client connection has been established and the appropriate persistent connection chosen, the two connections are spliced together. This requires the web switch to be modified at kernel level.

In ***one-way* architectures**, the server nodes send outbound packets directly to the client without them having to pass through the web switch.

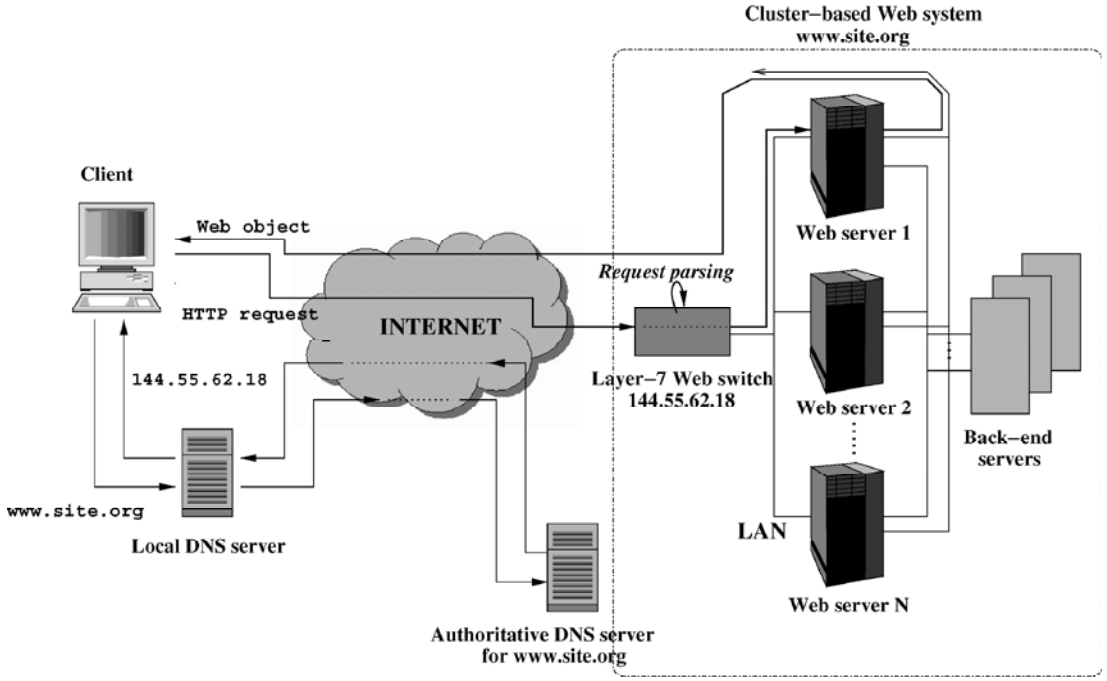

Figure 9 Layer-7 *one-way* architecture [1]

One-way architectures include:

- **TCP Connection hop**: A software based proprietary solution implemented by Resonate as a TCP based encapsulation protocol. Once the switch establishes a connection to the client and determines the target server, it hops the TCP connection to the server by encapsulating the IP packet in a Resonate Exchange Protocol (RPX) packet and sending it to the server. This operates at the network layer between the NIC and the TCP/IP stack, minimizing the latency of incoming packets. The server can reply directly to the client because servers in the pool share the same VIP address as the web switch.
- **TCP hand-off**: Once the switch establishes a connection to the client and determines the target server, it hands off its endpoint of the TCP

connection to the server, which can communicate directly with the client. Modification to the operating systems of both the web switch and servers is required. HTTP/1.1 connections are allowed by letting the web switch assign HTTP requests in the same connection to different target servers.

Dispatching Algorithms

In all types of web switch, dispatching algorithms are required in order to ensure that the load is shared between servers where multiple servers are eligible to receive a request. An overview of algorithms is presented below.

Because they run on the web switch, dispatching algorithms have access to all information the web switch has. As such they can be classified according to the type of web switch they are used on.

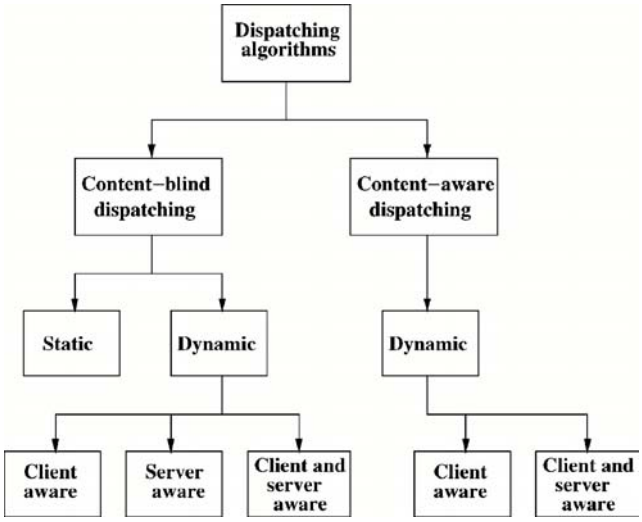

Figure 10 Taxonomy of dispatching policies in Web clusters [1]

Content-blind

Static algorithms do not consider any state information:

- **Random** – requests are distributed randomly with equal probability of each server
- **Round-Robin** – uses a circular list and distributes requests to each server in turn

These algorithms can be adapted to include weightings for servers with differing capacities.

Client state aware algorithms partition requests through client information such as IP address.

Server state aware algorithms generally use a server load index to assign requests:

- **Least Connections** assigns new requests to the server with fewest active connections
- **Fastest Response Time** assigns new requests to the server that is responding fastest, i.e. showing the smallest object latency time in the last observation interval
- **Dynamic weighted Round-Robin** is a weighted version of the Round-Robin algorithm that uses dynamic weights proportional to the server load index

Client and server state aware algorithms combine the above techniques in order to assign consecutive multiple connections from the same client to the same server, for example to ensure all requests from a SSL session are sent to the same server to avoid regenerating session keys etc

Content-aware

Client state aware content-aware algorithms use attributes of the request to partition the request:

- **URL Hashing** uses a hashing algorithm to partition the requests so only one server handles each request, this achieves the best cache hit rate but does not consider load balancing
- **Service Partitioning**: uses specialised servers for certain types of request
- **Size Interval Task Assignment with Equal load (SITA-E)** partitions content among the servers according to the size of the requested file, in order to separate servers for heavy tasks and light tasks (assuming the service time is proportional to the file size)
- **Client-Aware Policy (CAP)** recognises different requests use different resources on the server and divides requests into various classes according to which resources they use (E.g. disk-bound, CPU-bound and network-bound). CAP uses Round-Robin for each load class to share each load class among multiple servers.

Client and server state aware content-aware algorithms again combine client and server state information:

- **Locality-aware Request Distribution (LARD)** [12] considers both locality as with URL Hashing and load balancing. Requests for the same

web object are distributed to the same server node as long as it's load is below a given threshold, ensuring the object is more likely to be found in the disk cache of the server node. When the load increases above the threshold the request is assigned to the least loaded node, creating a pool of two servers likely to have this object in their cache. Subsequent requests between this pool, and the pool grows and shrinks automatically.

- **Cache Manager** relies on a cache manager aware of the cache content of all web servers, if a request is not in the cache it is assigned to the least loaded server. Otherwise the least loaded server with the object in its cache is selected (providing the load of that server is below a threshold)

2.4.3 Increasing Processing Speed

Instead of attempting to reduce the incoming request rate or increase the request processing capacity, high demand could be dealt with by changing the responses such that they can be generated more quickly. There does not appear to be any research using this methodology.

2.5 Summary

A great deal of research has been conducted into adapting web sites to cope with high demand. The vast majority of this research concentrates on increasing the web site's capacity to handle requests.

The aim of this project is to provide a certain level of increased capacity and fault tolerance, but to primarily concentrate on the novel technique of increasing speed at which requests are processed under high demand. This means that "everybody gets something" rather than "some people get everything", and may mean that during high loads the site seen by users is not as graphically intensive and aesthetically pleasing as usual, it is certainly more user-friendly than a "connection timed out" error message.

3 Design Overview

3.1 Introduction

This chapter provides a high-level overview of the design of the system that aims to meet the specification (See section 1.2).

3.2 Terminology

The following terminology will be used in the remainder of the report:

- **Virtual Server**: The virtual image of a single web server presented to the outside world by the cluster
- **Virtual Path/File**: A URL on the virtual web server
- **Pool Server**: A physical web server hosting some of the virtual paths on the virtual server
- **Physical Path/File**: A URL on one of the pool servers

3.3 System Architecture

The system is intended to provide a front-end web switch for a cluster-based web system with the following features (Defined in section 1.2):

- **Load balancing**
- **Heterogeneous server pool**
- **Throttling**
- **Adaptable**

Due to the requirements for the pool servers to be heterogeneous the switching will have to be content-aware, i.e. at layer-7 or application layer.

The architecture of a layer-7 web switch can be abstracted into two layers:

- The **control layer**, which makes the routing decisions by translating the virtual path of a request into one or more physical paths in order of preference.
- The **routing layer**, which accepts connections, extracts the virtual path from the request and queries the control layer for the appropriate virtual path(s). It then attempts to retrieve the response from a physical path starting with the most preferred. It then forwards the response to the client

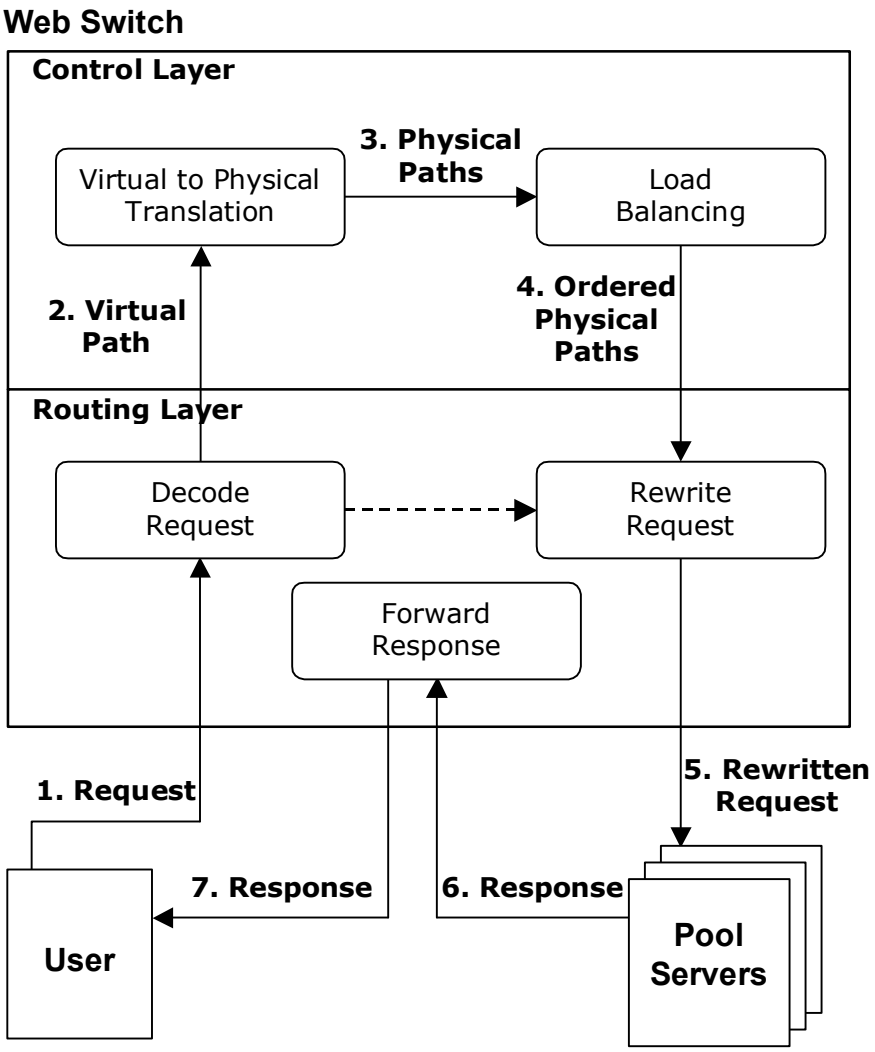

Figure 11 Simplified operation of a layer-7 web switch

To simplify the design this abstraction will be used and each layer will be tackled in turn.

3.4 Routing Layer

A large amount of research is available into routing mechanisms and the main techniques and methodologies are settled.[1] Hence existing software should be selected for this layer rather than trying to improve upon existing technologies. The selected software should be minimally modified to interface with the Control Layer.

3.5 Control Layer

The control layer is the "brain" of the web switch, and is where the features specified in the project's goals are implemented.

This layer can be seen to contain two major components, the Virtual File System and the Policy Engine. Additionally a configuration module handles loading and storing system configuration and initialises the Virtual File System and Policy Engine accordingly.

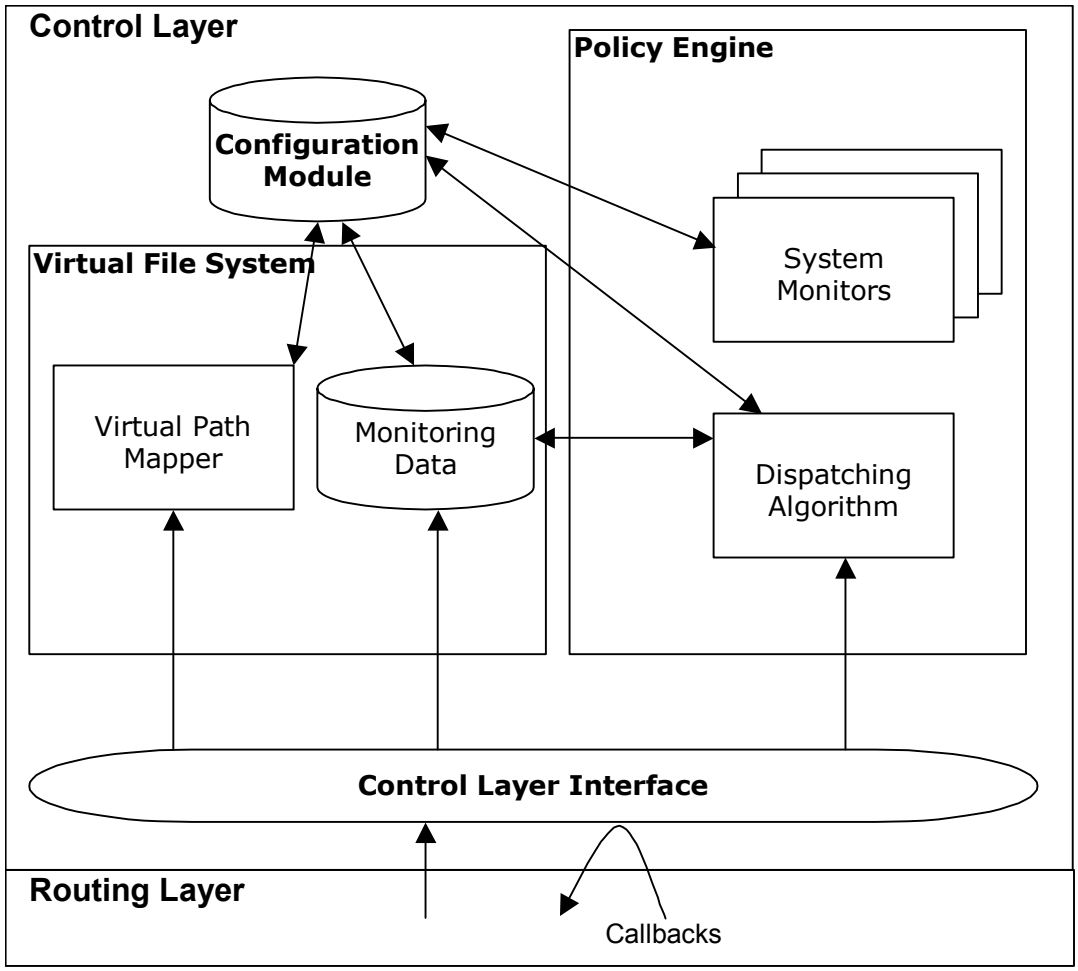

Figure 12 Control Layer structure

3.5.1 Control Layer Interface

Purpose

The layer interface is the only point of connection between the Control Layer and the Routing Layer.

Interface

The interface that the Control Layer exposes to the Routing Layer should take a virtual path and return an interface that allows iteration through all physical paths corresponding to the virtual path in order of preference. It should also provide callbacks to allow the Routing Layer to notify when connections are opened and aborted or finished to each physical path; along with timing data detailing how long a connection took.

Specification

The layer interface should retrieve an unordered list of all possible physical paths for the specified virtual path from the Virtual File System. It should then use the Policy Engine to sort these paths according to preference, and return the interface specified above allowing the routing layer to select a physical path and provide feedback.

3.5.2 Policy Engine

Purpose

The purpose of the policy engine is to cause different alternative physical paths (alternative versions of a given page) to be sent to the client in response to the requested virtual path. It also collects monitoring data on the system in order to make these decisions.

Interface

The interface that the Policy Engine exposes to the Layer Interface should take an unordered list of physical paths and return an ordered list of physical paths, sorted by preference.

Specification

The Policy Engine should prioritise physical paths based upon the following types of information:

- **Monitoring Data**
  - **System** Data
  - **Host** Data (about the pool server the Physical Path is on)
  - **Path** Data (about the Physical Path itself)
- **Metadata** (Data from the XML configuration file about the Physical Path to differentiate it from others)

The Policy Engine should be modular, allowing a new dispatching algorithm to be dropped in as a replacement for the default. Extra metadata should be storable in the Virtual File System and made available to the policy engine without modification to the Virtual File System, and it should be possible to plug in extra monitors to provide additional monitoring data.

3.5.3 Virtual File System

Purpose

The purpose of the Virtual File System is to map a virtual address to one or more physical addresses.

Interface

The interface provided by the Virtual File System should allow a virtual path to be translated into an unordered list of all possible physical paths containing the requested virtual file.

The Virtual File System should allow storage and retrieval of host and path data for the use of the Policy Engine. This should be achieved by each returned path providing an interface facilitating retrieval of monitoring data for the path and the host it is stored in. Each path's interface should allow real-time feedback of response times and connection loads from the Routing Layer.

Specification

The virtual to physical mapping should be specified by XML files either stored locally or on each web server. The XML format should allow individual files or entire directories to be mapped and should also allow additional data to be stored in order to be made available to the Policy Engine.

The in-memory representation of the data from the XML file should also allow monitoring data to be stored for the use of the Policy Engine.

The list of XML configuration files should be provided by the configuration framework, and should be parsed during initialisation. Additionally the system should support reloading of the configuration whilst the system is running. This should be supported as an atomic action to prevent inconsistent state.

3.5.4 Configuration Module

Purpose

The purpose of configuration module is to load and store configuration data for the control layer. It should also handle the initial setup of the layer, and plug dynamic components together, for example initialising the desired sorting algorithm and system monitors and plugging them into the policy engine.

Interface

The interface provided by the configuration module should allow components of the Control Layer to load and store configuration values required by them. It should also provide an Initialise function, which configures the Control Layer and prepares it for operation.

3.5.5 Summary

The Control Layer is broken up into a number of modular components, allowing each module to be developed and tested separately. Additionally the modularity allows publicly exposed interfaces to be defined for components that are designed to be replaceable by the end user.

The Control Layer as a whole uses the façade design pattern, where one interface class handles all incoming calls from other classes. Any objects returned through the façade class are converted to publicly exposed interfaces, limiting what methods can be accessed and keeping the interface as abstract as possible in order to reduce the constraints imposed on the internal structure of the Control Layer.

3.6 Initial Monitoring data and metadata

The policy engine specification lists the types of monitoring data that should be collected and states that the interface should be modular allowing additional monitoring data and metadata to be added with minimal code changes.

This section specifies an initial base set of monitoring data and metadata. In particular the response time monitoring data must be provided in the base set of monitoring data since it requires much tighter integration with the Virtual File System and Control Layer Interface.

The initial criteria for throttling were decided to be:

- **Bandwidth**: Although it is difficult to determine the bandwidth of the entire connection between client and server, it is possible to know the bandwidth of the connection between the web system and the Internet. Although the inbound bandwidth cannot be controlled easily, the outbound bandwidth can be controlled. Note that HTTP traffic involves an outbound response usually several times the size of the inbound request.

  The system should perform throttling to ensure that a web system on a

limited connection does not flood its outbound connection to the Internet when a large number of requests are being received. This ensures that all clients receive a response in a timely manner.

- **Load**: When a site with dynamic pages is under heavy load, requests take longer amounts of time and eventually take so long clients start timing out before the response is received. However the time each request takes varies, so some clients will receive responses and the majority will see timeout errors. This was seen in the recent sale of tickets for Glastonbury Festival, due to the extreme demand most people only received timeout errors despite trying several times. Some were able to receive the first booking page but then received a timeout after entering their credit card details. A lucky few managed to book tickets!

  The system should perform throttling to downgrade CPU/disk/database intensive dynamic pages to less intensive alternative versions under high load. This ensures that each request takes less time to service, and this means that more requests can be served in a given period of time, allowing all users to see something rather than some users see everything.

3.6.1 Monitoring data

Data collected

The initial monitoring data should provide sufficient data to perform throttling as described above:

- Current **system bandwidth usage**
- **Number of active connections to host** (pool server)
- Recent **host** (pool server) **standardised response times** (See standardising section below)
- Recent **path standardised response times** (See standardising section below)

Response times should be stored in a stack of fixed size; older response times should be removed to allow new ones to be added when the stack is full. Additionally response times should only be stored for a defined length of time before being expired in order to ensure data is relatively recent.

Rationale

System bandwidth usage is monitored in order to allow pages to be downgraded to low bandwidth versions in response to increasing load.

Number of active connections to host is monitored in order to allow a Least Loaded dispatching algorithm to balance load between servers where a tie exists.

Standardised response times are stored as a measure of system load: As load increases, limiting factors such as CPU power, disk access times and bandwidth start to cause responses to take longer. This makes response time a comprehensive and platform independent measure of end-to-end system performance rather than traditional methods requiring several statistics from various aspects of the system to be measured to try and gain an indication of performance.

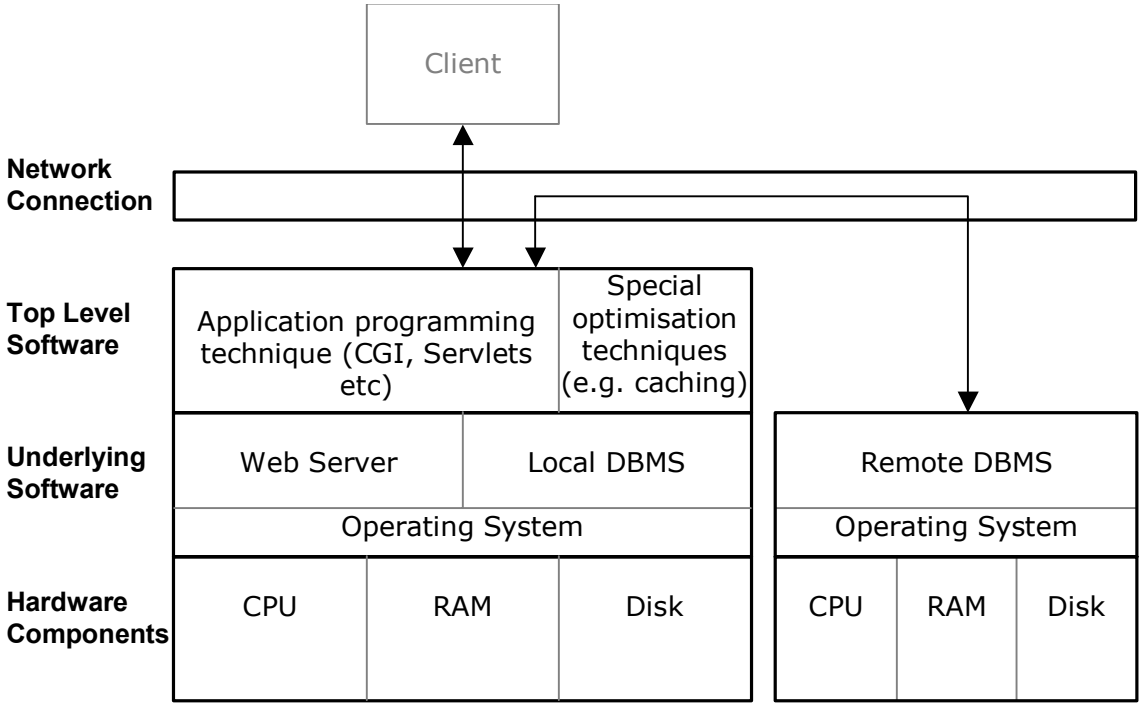

Figure 13 Factors determining web server performance (Based upon diagram from [19])

Research presented in [22] further confirms the suitability of response time as a measure of system loading. The authors construct a simulation to measure the response rate of a range of connections of varying bandwidth, with and without an ongoing Flash Crowd Event:

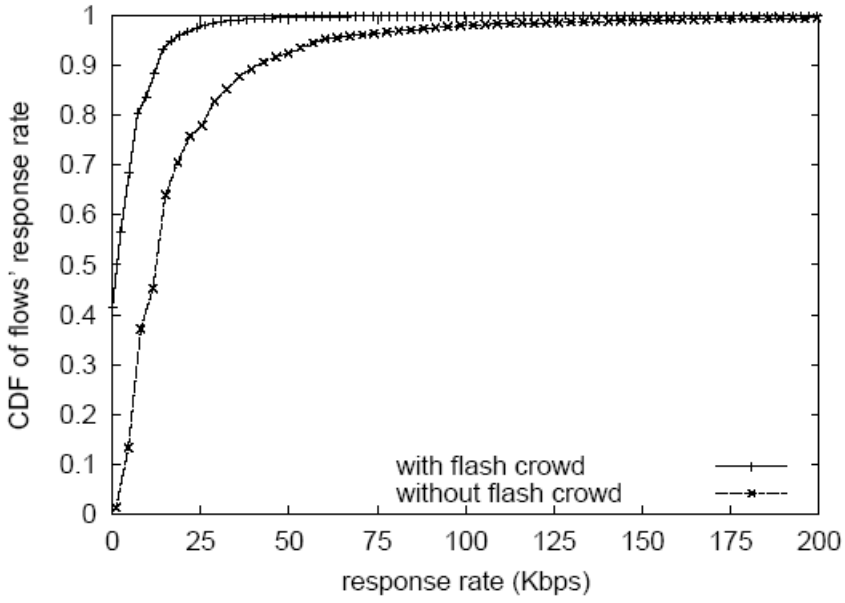

Figure 14 CDF of response rates during a Flash Crowd Event and during normal traffic

It can be seen that during a flash crowd event all response rates (And hence response times) drop. It can also be seen that this effect is seen much more markedly by high bandwidth connections (In lower bandwidth connections the client's connection starts to become a limiting factor). Since the connection between the web switch and pool server is high bandwidth, a significant change in response time should be shown under heavy load.

Advantages

The method of data collection should reduce the "herd effect" [10]. This caused by periodically testing the CPU usage or similar metrics from pool servers and directing traffic to the least loaded server. All requests are then sent to the same server until new information is propagated, quickly saturating the server.

In this design, because response times are being collected in real time and immediately fed back into the system, the average standardised response time provides an indication of the server's load at that time. Similarly the number of active connections to pool server is correct at any given time since this count is maintained by the system itself, which controls connections

However, note that this design assumes that the web system has a reasonable load upon pages since monitoring data is collected as requests are served. This has the advantage that all pages are incorporated in the monitoring rather than

an artificial test set. It also does not place any extra load upon the system by running tests. However it has the disadvantage that no statistics are gathered if the system is idle.

Because the measured response time is a thorough end-to-end indication of performance, it is affected by two main types of factors: (Note that network factors can be ignored because response times are measured only across the LAN between the pool server and web switch, so network bandwidth is assumed not to be a limiting factor)

- **Local factors** such as the pool server being heavily loaded
- **Global factors** usually specific to a particular type of page across all pool servers, for example a backend database server being heavily loaded causing CGI scripts to take longer than average to load.

By collecting recent standardised response times for each pool server in addition to each path, the system can compensate for local factors more effectively. A global factor will generally cause the response time average for all pool servers to rise equally and all pool servers will be balanced equally.

3.6.2 Metadata

The initial metadata provided describing each physical path should allow throttling based upon two of the major bottlenecks in web sites:

- **Bandwidth weighting**: The size of this version of the path relative to other versions - to allow throttling to prevent the outgoing connection from the cluster to the internet from being flooded, for example by switching to text-only pages rather than multimedia pages under high load
- **Load weighting**: The CPU usage/database load/complexity of this version of the path relative to other versions - to allow throttling to reduce the CPU/database load on the site, for example by switching from CGI pages to static pages under high load
- **Average & Standard Deviation of *target* response time**: See standardising section below

All metadata should be imported from the XML file along with the physical paths. Load and Bandwidth weightings should be specified as a value between 0 (minimum) and 1 (maximum).

3.7 Standardising Response Times

3.7.1 Introduction

Data of response times on its own is not useful in raw form since the typical response time varies between different paths depending on how CPU intensive they are, database access etc. Additionally when trying to ascertain a pool server's performance relative to targets although an average response time across all pages could be used, it would be more accurate if response times could be standardised so they can be easily combined into a response time for the pool server.

It seems sensible to assume that the response times under low load conditions to one particular page can be considered to be a random variable following a normal distribution $N(\mu, \sigma^2)$ since the majority of responses will be served in the average response time, some in slightly less time, some in slightly more time, but with no skew towards either side.

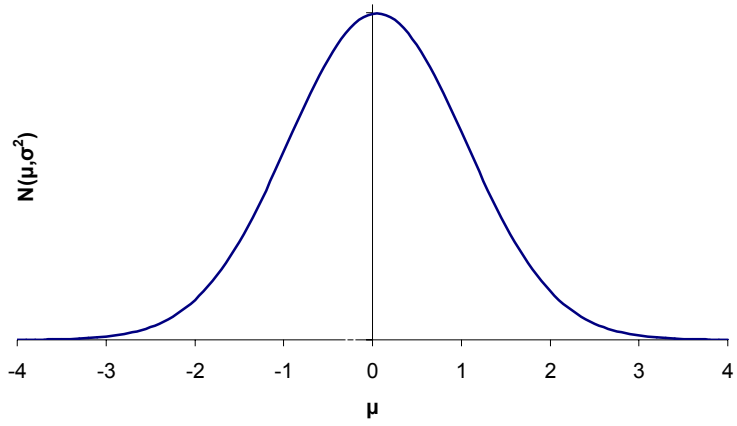

Figure 15 The standard normal distribution $N(0,1)$

3.7.2 Implementation

All measurements are taken over the local network from the web switch to a pool server. The following terminology is used:

- **Target Average/Standard Deviation**: The average and standard deviation of response times of each physical path over the local network under low load calculated *before using the system* (And independently of the system). These values are constant and are stored with each physical

path as its *target* response time average and standard deviation in the XML configuration file.

- **Measured Response Time**: The length of time taken by one particular attempt to load a given physical path.
- **Standardised Measured Response Time**: See below
- **Measured Average Response Time**: The average and standard deviation of *measured response times* of each physical path collected *whilst using the system*. These values vary constantly as the system runs.

Assuming the *measured response time* is a random variable following a normal distribution $N(\mu,\sigma^2)$ under low load, it can be then standardised (using the *target average/standard deviation*) to follow the standard normal distribution $N(0,1)$ giving a **Standardised Measured Response Time**:

$$x \leftarrow \frac{(x-\mu^2)}{\sigma}$$

All collected standardised response times now follow the standard normal distribution, so can be directly compared and averaged.

As load increases, limiting factors start to cause responses to take longer. This load increase causes the distribution of *measured response times* to skew towards the right hand side (See Figure 14). This implies that the average of the distribution will increase from the *target* average. An increase in the *measured* average compared to *target* average therefore implies the path is under heavy load. The significance of this increase can be judged by comparing the difference to the *target* standard deviation.

If instead of averaging *measured* response times, the average of *standardised measured* response times is taken, under low load this will be 0. Under increasing load, the same effect as above is noticed, but is relative to 0 rather than the *target* average so a positive value implies requests are taking longer than average. Because the scale has been standardised according to the standard deviation of response times, a value of +1 means the average response time is 1 standard deviation above the mean and so on. Hence the **average standardised measured response time** can be compared to other values for different files or other pool servers.

3.8 Summary

An architecture has been described which allows a pool of web servers to be presented to the outside world as a single virtual server.

The Control Layer Interface ensures the Control Layer presents a suitable façade to the Routing Layer, ensuring the Control Layer can be retrofitted to any desired Routing Layer.

The Virtual File System component allows fully controllable mappings between paths on the virtual server and paths on the pool servers, enabling the pool to be heterogeneous. It also allows metadata to be provided with paths and monitoring data to be collected on their performance.

The Configuration Module allows filters to be plugged into the system to monitor performance.

The Policy Engine gathers all system state information together and presents it to a dispatching algorithm, which can easily be replaced by a user-defined algorithm if desired. A sensible base set of monitoring data has been specified.

4 Detailed Design

4.1 Introduction

This chapter builds upon the high-level design overview (See section 3) to provide a detailed design of the system. It also details the tools used to implement the system, the rationale behind design decisions that were taken and the problems that arose during implementation.

The interface between each module of the system is described by listing component classes of each module. A format similar to *javadoc* is used; only public methods are specified. An outlined Method Detail section describes the internal operation of methods with non-obvious side effects or implementing noteworthy algorithms.

4.2 Implementation Tools and Techniques

This section provides an overview of the tools and techniques that should be used in implementing the system.

Java 2 Platform, Standard Edition (J2SE 1.4.2)

The Java 2 platform [5] was chosen for development of the main application due to the wide range of feature-rich APIs it provides. These allow an application to be quickly and easily created using pre-written and pre-tested APIs for more complex features.

Although Java runs more slowly than native code due to the overhead of its Virtual Machine, significant advantages are gained through rapid development capabilities and portability. Additionally it is generally accepted that Java is very suitable for network applications since these tend to be IO bound rather than CPU bound, and as such the Java code running slightly slower has a negligible effect.

Eclipse

Eclipse 3.0 [6] was chosen as the development platform due to its great suitability as an IDE for Java development and the extensive range of features it provides to make Java development as quick and simple as possible.

CVS

Concurrent Versions System[7] was chosen for source code control, providing an additional backup of code in addition to a full revision history allowing changes to be reverted without requiring code to be rewritten.

Coding Standards

Sun's Java coding style[8] was chosen as the standard for source code formatting to ensure all code is presented in a consistent format.

Particular attention was paid to the organisation of packages, building up a namespace to group related classes together. The project was designated "mercury" and all new classes specific to the project were created in appropriately named sub packages of the mercury package.

4.3 Control Layer

4.3.1 XML Configuration Format

The DTD of the XML configuration file is shown below. Additional attributes can be added to the <file> tag if desired, these will be made available to the dispatching algorithm without any code changes.

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

<!ELEMENT servermanifest (virtualfile*)>

<!ELEMENT virtualfile (file*)>
<!-- Virtual Path -->
<!ATTLIST virtualfile path CDATA #REQUIRED>

<!ELEMENT file (#CDATA)>
<!-- URL of physical path  -->
<!ATTLIST file url CDATA #REQUIRED>
<!-- Bandwidth weighting -->
<!ATTLIST file bandwidth CDATA #DEFAULT "0.5">
<!-- Load weighting -->
<!ATTLIST file load CDATA #DEFAULT "0.5">
<!-- Target avg response time (ms) -->
<!ATTLIST file average CDATA #DEFAULT "5">
<!-- Target response time std deviation (ms) -->
<!ATTLIST file stdev CDATA #DEFAULT "1">
```

The state machine for the parser is shown below:

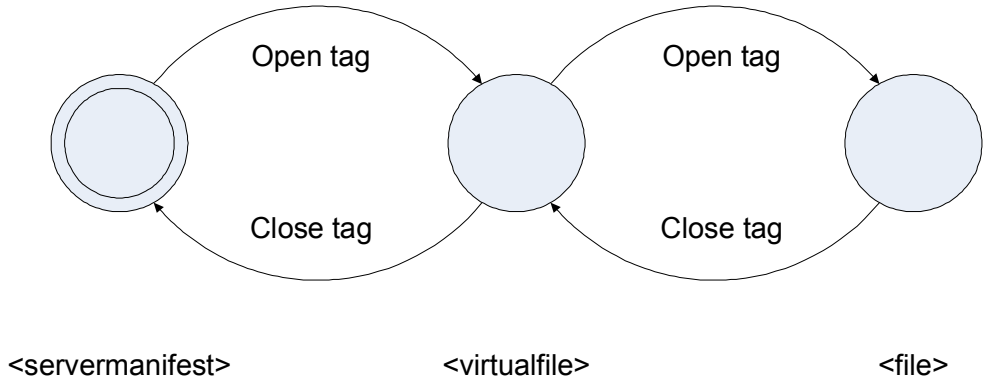

Figure 16 XML Parser state machine

A typical configuration file might look like the following:

```
<servermanifest>
 <virtualfile path="/html/">
  <file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync20:7962/HI/html/"/>
  <file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync21:7962/HI/html/"/>
  <file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync20:7962/MED/html/"/>
  <file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync21:7962/MED/html/"/>
  <file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync20:7962/LOW/html/"/>
  <file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync21:7962/LOW/html/"/>
 </virtualfile>

 <virtualfile path="/cgi-bin/">
  <file load="1" bandwidth="1" average="6" stdev="83"
url="http://sync21:7962/HI/cgi-bin"/>
  <file load="0.5" bandwidth="1" average="6" stdev="83"
url="http://sync20:7962/MED/cgi-bin"/>
  <file load="0.5" bandwidth="1" average="6" stdev="83"
url="http://sync21:7962/MED/cgi-bin"/>
  <file load="0" bandwidth="1" average="6" stdev="83"
url="http://sync20:7962/LOW/cgi-bin"/>
 </virtualfile>
</servermanifest>
```

4.3.2 Control Layer Interface (mercury.urimapper)

The control layer interface provides the interface between the routing layer and the control layer. The URIMapper class provides this interface (And hence exhibits the façade design pattern).

public class mercury.urimapper.URIMapper

An instance of this class is created for each request.

`public` **`URIMapper`**`(java.lang.String strURI) throws URIMapperException`

Constructor. Takes the requested virtual path as a parameter. Constructs a new *URIMapper* object for the requested virtual path. If an error occurs, an *URIMapperException* is thrown.

Method Detail: Retrieves a list of Physical Paths containing the virtual path from the Virtual File System. It then calls the selected dispatching algorithm to sort the list of paths in order of priority and stores this internally.

`public void` **`moveStart`**`()`

Selects the first Physical Path in the list.

`public void` **`moveNext`**`()`

Selects the next Physical Path in the list.

`public boolean` **`hasMore`**`()`

Returns *true* if more Physical Paths are available in the list.

`public java.lang.String` **`CurrentURI`**`()`

Returns the URL of the currently selected Physical Path.

`public MappedURICallback` **`getCurrentURINotifyCallback`**`()`

Returns a *MappedURICallback* interface for the currently selected Physical Path.

public class mercury.urimapper.URIMapperException

Extends java.lang.Exception. Thrown when an error occurs creating an URIMapper object. It contains a description of the error and a HTTP response code allowing a response to be sent to the client (E.g. a 404 File Not Found might be thrown if no physical paths are found for the requested virtual path)

`public` **`URIMapperException`**`(java.lang.String strStatusCode, java.lang.String errorMessage)`

Constructor. Constructs an *URIMapperException* with the specified HTTP status line (E.g. "404 Not Found") and message (A description of the error)

`public java.lang.String` **`getHttpStatusCode`**`()`

Returns the status line associated with the exception.

`public java.lang.String` **`getErrorMessage`**`()`

Returns a description of the exception.

public interface mercury.urimapper.MappedURICallback

This is the callback interface and allows the Routing Layer to provide monitoring data feedback about a Physical Path

`public void` **`notifyComplete`**`(long timeMilliseconds)`

Called when a request has been successfully completed, specifies the time taken in milliseconds

`public void` **`notifyActive`**`()`

Called when a request is started. The programmer must ensure either *notifyComplete* or *notifyAbort* is called exactly once after each *notifyActive* call

`public void` **`notifyAbort`**`()`

Called when a request has been aborted

4.3.3 Policy Engine (mercury.logic)

public interface mercury.logic.Balancer

This is the interface all dispatching algorithms must implement.

`public java.util.List` **`balance`**`(java.util.Collection uris)`

Receives an unordered Collection of *MappedURI* objects. It should return an ordered list containing the same objects listed in order of preference.

`public void` **`dumpState`**`(java.io.PrintStream out)`

Dumps the current state of the dispatching algorithm to the specified *PrintStream*.

`public void` **`StartBalancer`**`(MercuryConfig config)`

Called to initialise the dispatching algorithm. Receives a reference to the system *MercuryConfig* object (Allowing configuration data to be loaded/saved, and also giving access to configured system monitoring data)

`public void` **`StopBalancer`**`(MercuryConfig config)`

Called to shutdown the dispatching algorithm. Receives a reference to the system *MercuryConfig* object (Allowing configuration data to be saved)

public interface mercury.logic.MappedURI

Provides the dispatching algorithm with access to all monitoring data and metadata about a Physical Path

`public java.lang.String` **`getURL`**`()`

Returns the URL of this Physical Path

`public double` **`getBandwidth`**`()`

Returns the bandwidth weighting of this Physical Path

`public double` **`getLoad`**`()`

Returns the load weighting of this Physical Path

`public java.lang.String` **`getAttribute`**`(java.lang.String name)`

Returns an additional attribute from the XML <file> tag or *null* if the attribute was not specified for this Physical Path.

`public MappedHost` **`getHost`**`()`

Returns a *MappedHost* interface representing the host that this Physical Path resides on

`public double` **`getAvgResponse`**`()`

Returns the current *average standardised measured response time* for this Physical Path

public interface mercury.logic.MappedHost

Provides the dispatching algorithm with access to all monitoring data about the host a physical path resides on

`public long` **`getConnectionCount`**`()`

Returns the number of connections currently active on the host

`public double` **`getConnectionFraction`**`()`

Returns the fraction of total connections that are on this host, i.e. 1 if all connections are on this host, 0 if no connections are on this host.

`public double` **`getAvgResponse`**`()`

Returns the current *average standardised measured response time* for this host

`public java.lang.String` **`getName`**`()`

Returns the hostname of this host

public class mercury.logic.DefaultBalancer

Implements Balancer. This class implements the default dispatching algorithm, described in section 4.3.8.

4.3.4 Policy Engine System Monitors (mercury.monitors)

public interface mercury.monitors.SystemMonitor

Interface. This is the interface all system monitors must implement.

`public java.lang.String` **`GetName`**`()`

Returns the name of this monitor

`public void` **`StartMonitor`**`(MercuryConfig config)`

Called to initialise the monitor. Receives a reference to the system *MercuryConfig* object (Allowing configuration data to be loaded/saved)

`public void` **`dumpState`**`(java.io.PrintStream out)`

Dumps the current state of the monitor to the specified *PrintStream*.

`public void` **`StopMonitor`**`(MercuryConfig config)`

Called to shutdown the monitor. Receives a reference to the system *MercuryConfig* object (Allowing configuration data to be loaded/saved)

public interface mercury.monitors.SystemBandwidthMonitor

Extends SystemMonitor. This is the interface the system bandwidth monitor must implement. (The code of the monitor is specific to the routing layer)

`public double` **`GetCurrentBandwidth`**`()`

Returns the current bandwidth usage of the system in bytes per second.

4.3.5 Virtual File System (mercury.vfs)

abstract class mercury.vfs.VirtualPath

Abstract. This class is the in-memory representation of the virtual file system. Each instance represents a node in the VFS tree. Not visible outside of the package.

`public void` **`addPath`**`(RealPhysicalPath p)`

Adds the selected *RealPhysicalPath* to the list of paths this Virtual Path maps to (Used by the XML importer)

public mercury.vfs.VirtualPathRoot

Extends VirtualPath. This class is the in-memory representation of the root of the virtual file system tree. It exhibits a variant of the singleton design pattern, many instances may be created but only one may be currently active. This allows atomic loading of the configuration.

`public` **`VirtualPathRoot`**`()`

Constructor. Constructs a new *VirtualPathRoot* object representing the root of an empty VFS tree. This allows configuration to be loaded on this tree, which can then be made active as an atomic operation

`public static VirtualPathRoot` **`getCurrentRoot`**`() throws mercury.vfs.VFSException`

Returns the root of the currently active VFS tree. Throws a *VFSException* if no tree is currently active

`public java.util.Collection` **`getPhysicalPaths`**`(java.lang.String URI) throws mercury.vfs.VFSException`

Returns a collection of *PhysicalPath* objects representing all Physical Paths that the given URI (Virtual Path) maps to

Method Detail: Calls a recursive method provided by the *VirtualPath* class, which recurses down the tree from the root until the specified node is reached. During this recursion *PhysicalPath* objects associated with each node are collected by calling the associated RealPhysicalPath's GeneratePath*(subPath)* method with the remainder of the URI string see *RealPhysicalPath.GeneratePath* for more detail.

This means that entire directories can be mapped and specific mappings can be added for specific subdirectories or files – for example one *PhysicalPath* could be specified for the root path, implying one server has a copy of all files. Subpaths could then add to this mapping by defining additional servers for specific files or directories.

`public VirtualPath` **`getVirtualPath`**`(java.lang.String URI)`

Returns a *VirtualPath* object representing the node of the VFS tree corresponding to the given URI. The specified node and any parent nodes are created as necessary

Method Detail: Calls a recursive method provided by the *VirtualPath* class, which recurses down the tree from the root until the specified node is reached, creating nodes as required

`public void` **`DumpTree`**`(java.io.PrintStream out)`

Dumps a string representation of the entire tree to the specified *PrintStream*

Method Detail: Calls a recursive method provided by the *VirtualPath* class which recurses down the tree printing nodes

`public void` **`dumpHosts`**`(java.io.PrintStream out)`

Dumps the current state of all hosts to the specified *PrintStream*.

`public void` **`loadConfig`**`(java.lang.String url)`

Loads an XML config file from the specified URL and adds all information to this VFS tree

Method Detail: Creates a new *XMLLoader* class to load configuration into this tree. It then uses SAX to parse the configuration file using the *XMLLoader* class as a handler. SAX [28] was chosen for speed because it operates in one pass, rather than DOM [21] which uses two passes and requires the entire XML file to be parsed into memory before it can be used.

`public void` **`makeActiveRoot`**`()`

Makes this VFS root the active VFS tree. This allows the a new configuration to be loaded in isolation and then switched into service atomically

class mercury.vfs.VirtualPathNode

Extends VirtualPath. This class is the in-memory representation of a node of the virtual file system. Not visible outside of the package.

`public java.lang.String` **`getName`**`()`

Returns the name of this node (directory/file name)

class mercury.vfs.XMLLoader

Extends org.xml.sax.helpers.DefaultHandler. This class is used as a handler by SAX when parsing XML config files. Not visible outside of the package.

`public` **`XMLLoader`**`(mercury.vfs.VirtualPathRoot root)`

Constructor. Constructs a new XMLLoader object ready to parse config files and add the results to the VFS root specified

```
public void startElement(java.lang.String uri,
java.lang.String localName, java.lang.String qName,
org.xml.sax.Attributes attributes) throws org.xml.sax.SAXException
```

Handles an opening tag

```
public void endElement(java.lang.String uri,
java.lang.String localName, java.lang.String qName) throws
org.xml.sax.SAXException
```

Handles a closing tag

public class mercury.vfs.VFSException

Extends java.lang.Exception. This class represents a VFS error.

```
public VFSException(java.lang.String message)
```

Constructor. Constructs a new *VFSException*.

class mercury.vfs.PhysicalHost

Implements MappedHost. This class represents a Physical Host (Pool server). Not visible outside of the package.

```
public static PhysicalHost CreateHost(java.lang.String name)
```

Returns a *PhysicalHost* object representing the specified hostname. A new object is created if necessary

Method Detail: Monitoring data for physical hosts is preserved across reloads of the VFS. This is achieved by registering all hosts in a private static *Map* object. This method checks to see if the host is in the map, if it is not a new host is created and added to the map.

```
public java.lang.String toString()
```

Returns a string detailing the current state of the host and the values of monitoring data

```
public long getConnectionCount()
```

Returns the current active connection count of this host.

```
public double getConnectionFraction()
```

Returns the fraction of total connections that are on this host, i.e. 1 if all connections are on this host, 0 if no connections are on this host.

```
public void notifyActive()
```

Increments the active connection count of this host (Synchronized for thread compatibility)

```
public void notifyAbort()
```

Decrements the active connection count of this host (Synchronized for thread compatibility)

`public void` **`notifyComplete`**`(double normalisedTime)`

Decrements the active connection count of the host. Records the *normalised measured response time* in the host's monitoring data.

> **Method Detail**: A synchronized method decrements the active connection count of the host. normalisedTime is then added to the host's *StatsArray*.

`public double` **`getAvgResponse`**`()`

Returns the *average normalised measured response time* for this host.

> **Method Detail:** Retrieves the value from the host's *StatsArray*

`public java.lang.String` **`getName`**`()`

Returns the hostname of this host.

public abstract class mercury.vfs.PhysicalPath

Implements MappedURI, MappedURICallback. This class represents a Physical Path.

class mercury.vfs.RealPhysicalPath

Extends RealPhysicalPath. This class represents an uncloned Physical Path. Not visible outside of the package.

`public PhysicalPath` **`GeneratePath`**`(java.lang.String subPath)`

Called by *VirtualPath* when mapping virtual paths to physical paths. This returns a *PhysicalPath* object representing the physical path with the subPath, if any, appended to the URL (Allows mapping of entire directories by appending filenames/subdirectories to the URL)

> **Method Detail**: If the subPath is empty, GeneratePath simply returns *this* (the *RealPhysicalPath* object it was called on). Otherwise a new *ClonedPhysicalPath* is created and returned, with *this* as its parent and the specified subPath.

```
public static RealPhysicalPath CreatePath(java.lang.String url,
double bandwidth, double load, double responseAvg,
double responseStdev, java.util.Map attributes) throws
java.net.MalformedURLException
```

Called by XML importer. Returns a *RealPhysicalPath* object representing the specified URL. A new object is created if necessary; otherwise the existing one is updated with the parameters specified. Throws a *MalformedURLException* if the URL could not be parsed.

Method Detail: Monitoring data for physical paths is preserved across reloads of the VFS. This is achieved by registering all paths in a private static *Map* object. This method checks to see if the path is in the map, if it is not a new path is created and added to the map. Additionally note that another Map object is passed in containing any extra tag attributes found in the XML file.

If a new path is created, the method also uses *java.net.URL* to extract the hostname from the URL and calls *PhysicalHost.CreateHost* to obtain a *PhysicalHost* object for the pool server containing the physical path, which is stored in the new *RealPhysicalPath* object.

`public java.lang.String` **`getURL`**`()`

Returns the URL of this Physical Path

`public double` **`getBandwidth`**`()`

Returns the bandwidth weighting of this Physical Path

`public double` **`getLoad`**`()`

Returns the load weighting of this Physical Path

`public java.lang.String` **`getAttribute`**`(java.lang.String name)`

Returns an additional attribute from the XML <file> tag or *null* if the attribute was not specified for this Physical Path.

Method Detail: Retrieves the specified attribute from a private Map object containing extra attributes found in the XML tag

`public MappedHost` **`getHost`**`()`

Returns a *MappedHost* interface representing the host that this Physical Path resides on

`public double` **`getAvgResponse`**`()`

Returns the current *average standardised measured response time* for this Physical Path

Method Detail: Retrieves the value from the path's *StatsArray*

`public void` **`notifyComplete`**`(long timeMilliseconds)`

Normalises the given time and records this *normalised measured response time* in the path's monitoring data. Also passes this information to the *PhysicalHost* associated with the path.

Method Detail: Normalises timeMilliseconds according to the path's *target average/standard deviation*. Adds this to the path's stats array and calls the associated *PhysicalHost*'s notifyComplete(normalisedTime) method

`public void` **`notifyActive`**`()`

Called when a request is started. The programmer must ensure either *notifyComplete* or *notifyAbort* is called exactly once after each *notifyActive* call.

Method Detail: Calls the associated *PhysicalHost*'s notifyActive() method

`public void` **`notifyAbort`**`()`

Called when a request has been aborted

Method Detail: Calls the associated *PhysicalHost*'s notifyAbort() method

`public java.lang.String` **`toString`**`()`

Returns a string detailing the current state of the path and the values of monitoring data

class mercury.vfs.ClonedPhysicalPath

Extends RealPhysicalPath. This class represents a cloned Physical Path. Not visible outside of the package.

`ClonedPhysicalPath``(RealPhysicalPath parent, java.lang.String subPath)`

Constructs a new *ClonedPhysicalPath* object with the specified parent and subpath relative to the parent's path.

`public java.lang.String` **`getURL`**`()`

Returns the URL of this Physical Path

Method Detail: Returns the parent's URL with the subpath appended to it

`public double` **`getBandwidth`**`()`

Calls the parent's *getBandwidth()* function and returns the value

`public double` **`getLoad`**`()`

Calls the parent's *getLoad()* function and returns the value

`public java.lang.String` **`getAttribute`**`(java.lang.String name)`

Calls the parent's *getAttribute(name)* function and returns the value

`public MappedHost` **`getHost`**`()`

Calls the parent's *getHost()* function and returns the value

`public double` **`getAvgResponse`**`()`

Calls the parent's *getAvgResponse()* function and returns the value

`public void` **`notifyComplete`**`(long timeMilliseconds)`

Calls the parent's *notifyComplete(timeMiliseconds)* function and returns the value

`public void` **`notifyActive`**`()`

Calls the parent's *notifyActive()* function and returns the value

`public void` **`notifyAbort`**`()`

Calls the parent's *notifyAbort()* function and returns the value

`public java.lang.String` **`toString`**`()`

Calls the parent's *toString()* function and returns the value of this appended to the cloned URL

class mercury.vfs.StatsArray

This class provides a fixed size stack, values are pushed on and older values are pushed off to make room if necessary. Values are expired after a given time. Not visible outside of the package.

`public` **`StatsArray`**`(int maxCapacity, long maxAge)`

Creates a new statistics array with the specified capacity and maximum age

`public void` **`add`**`(double normalisedValue)`

Adds the specified normalised value to the array (Synchronized for thread safety)

Method Detail: First the size of the array is checked, and the last element removed if it is at capacity. The new value is added as the first element of the array.

Secondly the cached average is recalculated:
The expiry time of the last element of the array is checked and it is removed if it has expired. This is repeated until the list is empty or the checked element has not expired.
If the array is empty the average is set to zero and the last recalculation time is set to the current time.
If the array is not empty the average is set to the sum of the array values divided by the number of values and the last recalculation time is set to the current time.

The cached average is used to speed up client requests by not having to recalculate the data. It is calculated after adding a value because at this point the proxy is in a cleanup phase and the client has received their data. This means add can be called without slowing down the client response.

`public double` **`getAverage`**`()`

Returns the cached average of the values currently stored in the array (Synchronized for thread safety)

Method Detail: To ensure the cached average is up to date if add() has not been called recently, the last recalculation time is checked, if this was more than 5 minutes ago, the recalculation procedure described in the add() procedure above is run to expire old statistics and recalculate the average.

4.3.6 Configuration Module (mercury.config)

public interface mercury.config.ConfigStore

Interface. This is the interface the system configuration store class must implement.

```
public java.util.Properties
getProperties(java.lang.String sectionName)
```

Returns all properties for the specified sectionName (sectionName should be the fully qualified name of the calling class)

```
public void setProperties(java.lang.String sectionName,
java.util.Properties prop)
```

Sets the properties for the specified sectionName (sectionName should be the fully qualified name of the calling class)

`public java.lang.String` **`getProperty`**`(java.lang.String section, java.lang.String key)`

Returns the property named key for the specified section (section should be the fully qualified name of the calling class), or null if it is unset

`public java.lang.String` **`getProperty`**`(java.lang.String section, java.lang.String key, java.lang.String defaultstring)`

Returns the property named key for the specified section (section should be the fully qualified name of the calling class), or defaultstring if it is unset

`public void` **`setProperty`**`(java.lang.String section, java.lang.String key, java.lang.String value)`

Sets the property named key to value for the specified section (section should be the fully qualified name of the calling class)

public class mercury.config.MercuryConfig

This class handles the initialisation of the Control Layer, and creates all necessary objects. It also connects plug in components such as the dispatching algorithm and system monitors. It uses a modified version of the singleton design pattern (only one instance can be active, but several can be created)

`public MercuryConfig (ConfigStore store)`

Constructor. Constructs a new *MercuryConfig* object, using the specified *ConfigStore* to load/save configuration values. Throws an *Error* if *Start()* has already been called on a *MercuryConfig* object (A configuration instance has been activated).

Method Detail: This method constructs the object only. This allows *setSystemMonitor* to be called to set up monitors before then calling *Start()*. This ensures all monitors have been initialised when the system dispatching algorithm starts.

`public static final java.lang.String` **`SYSMONITOR_OUTBOUNDBANDWIDTH`**

The name of the system outbound bandwidth monitor, to be used with *get/setSysMonitor*

`public void` **`dumpMonitors`**`(java.io.PrintStream out)`

Dumps the current state of all monitors to the specified *PrintStream*.

Method Detail: Calls the *dumpState()* method of all registered monitors.

`public ConfigStore` **`getStore`**`()`

Returns the *ConfigStore* object to be used for loading/saving configuration data

`public static MercuryConfig` **`Start`**`()`

Starts the Control Layer, loads configuration values and starts all monitors. Throws an *Error* if *Start()* has already been called on a *MercuryConfig* object (A configuration instance has been activated).

Method Detail: This instance is saved in a private static variable representing the currently active instance. A list of VFS configuration files is retrieved from the *ConfigStore* and the VFS system is initialised. Then the name of the selected dispatching algorithm is retrieved from the *ConfigStore* class. A new instance of this class is created using the Java Reflection API, and its *StartBalancer(config)* method is called to initialise it.

`public static void` **`Shutdown`**`()`

Shuts down the Control Layer.

Method Detail: *StopBalancer(config)* is called to instruct the selected dispatching algorithm to save it's configuration and shut down. Then *StopMonitor(config)* is similarly called on all registered system monitor classes. Finally the *MercuryConfig* instance is deleted.

`public void` **`setSysMonitor`**`(java.lang.String name,`
`SystemMonitor monitor)`

Registers the *SystemMonitor* object passed with the specified name. Throws an *Error* if *Start()* has already been called on a *MercuryConfig* object (A configuration instance has been activated).

Method Detail: The *SystemMonitor* is added to a private *Map* object with the name as the key. The *StartMonitor(config)* of the monitor is then called to allow it to load configuration data.

`public static MercuryConfig` **`getInstance`**`()`

Returns the current instance of *MercuryConfig*. Throws an *Error* if *Initialise()* has not been called

`public SystemMonitor` **`getSysMonitor`**`(java.lang.String name)`

Returns the named system monitor or throws an *Error* if *setSysMonitor* has not been called to register the monitor

Method Detail: Retrieves the *SystemMonitor* object from the private *Map* containing registered monitors

`public Balancer` **`getSysBalancer`**`()`

Gets the selected dispatching algorithm

`public int` **`getHostStatsMax`**`()`

Returns the configured maximum number of *normalised measured response times* to store with each *PhysicalHost*

`public long` **`getHostStatsAge`**`()`

Returns the configured maximum age of *normalised measured response times* stored with each *PhysicalHost*

`public int` **`getPathStatsMax`**`()`

Returns the configured maximum number of *normalised measured response times* to store with each *PhysicalPath*

`public long` **`getPathStatsAge`**`()`

Returns the configured maximum age of *normalised measured response times* stored with each *PhysicalPath*

4.3.7 Debugging and logging (mercury.debug)

`public class SystemTrace`

Extends Thread. Allows a thread to be created which runs in the background and periodically logs the state of the Control Layer to a file. Uses the singleton design pattern (Only one thread can ever be active)

`public static SystemTrace` **`StartThread`**`(int delaySeconds, java.lang.String filePath)`

Returns a the currently running thread object, creating one if necessary. The thread's output path and delay is set to the specified values. If filePath is "GUI", statistics are displayed in a Swing GUI window instead of written to file.

`public void` **`run`**`()`

Loops and periodically dumps the state of the Control Layer to a file.

> **Method Detail**: Appends the current time in milliseconds to the file, then calls *MercuryConfig.dumpMonitors()* to dump all system monitors, *MercuryConfig.getSysBalancer().dumpState()* to dump the dispatching algorithm's state, and *VirtualPathRoot.getCurrentRoot().DumpTree()* to dump the virtual file system tree.

4.3.8 Initial Dispatching Algorithm

Initially, the **system bandwidth throttle** is be calculated:

$$T_B = \sqrt{\min\left(\max\left(\frac{\left(\frac{R_{current}}{R_{\max}} - 0.8\right)}{0.2}, 0\right), 1\right)}$$

Where $R_{current}$ is the current outgoing bandwidth rate and R_{max} is the bandwidth rate cap. This is designed to keep the bandwidth throttle at 1 (unrestricted) until the outgoing bandwidth reaches 80% of the cap value. The throttle value then begins to drop until it reaches 0 at 100% of the cap value. It also drops increasingly quickly as 100% is approached.

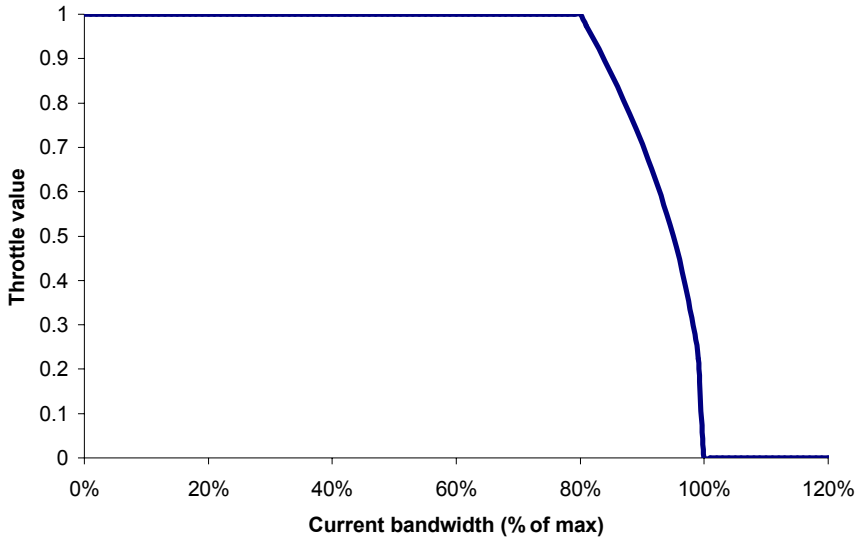

Figure 17 System bandwidth throttle

The **path bandwidth ranking** (R_B) is computed for each path:

$$R_B = \begin{cases} (-W_B + T_B) + (1 - T_B) & \text{if } W_B > T_B \\ W_B + (1 - T_B) & \text{otherwise} \end{cases}$$

Where T_B is the system bandwidth throttle as previously specified, and W_B is the path's bandwidth weighting (from metadata). This is designed to rank paths between 1 and 0 such that they are prioritised as follows: Firstly paths below the bandwidth throttle, in descending order of bandwidth weighting, and secondly paths above the bandwidth throttle in ascending order of bandwidth weighting.

This ensures paths below the throttle are prioritised over paths that are above the throttle, and secondarily paths are prioritised in ascending order of distance from the throttle value.

Also the **path load ranking** (R_L) is computed for each path:

$$R_L = W_L \times (1 - \mu)$$

Where W_L is the path's load weighting (from metadata) and μ is the *average standardised measured response time* for the path. This is designed to reward or punish paths as they perform above or below 1 standard deviation of their average response time. Paths are rewarded/punished increasingly heavily if they have a high load weighting and also if they perform significantly above or below 1 s.d. Paths with a load weighting of 0 are neither rewarded nor punished. This is similar to a utility value and represents the usefulness of the path to the user given its current responsiveness.

The reason 1 standard deviation was chosen is because according to the standard normal distribution (which μ follows – see standardising section below) 84% of response times will be below this value if they conform to the standard normal distribution, i.e. if the server is lightly loaded. As load increases, standardised measured response times will start to increase above 1 s.d and this will pull μ above 1 s.d.

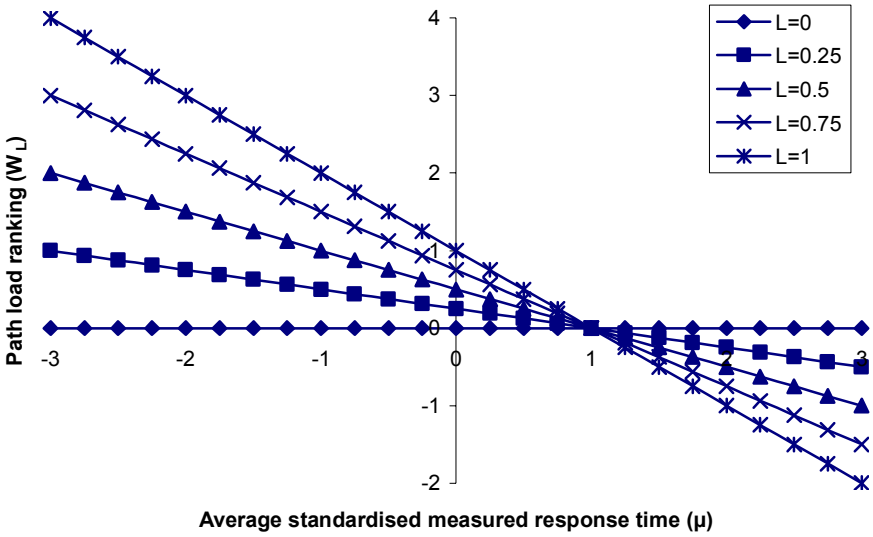

Figure 18 Path load ranking

The **host load ranking** (R_H) is computed for each host as follows:

$$R_H = 1 - \frac{L_H}{L_{Total}}$$

Where L_H is the number of active connections to this host and L_{Total} is the number of active connections to all hosts.

A weighted sum of the following values is then computed for each path (weights should be user-definable to allow different priorities to be set for different requirements):

- Path bandwidth ranking (R_B)
- Path load ranking (R_L) (Normalised to be between 0 and 1)

Paths should then be sorted by this weighting in descending order. If two paths have an equal weighting they should then be sorted by:

- Ascending sort by *average standardised measured response time* for host
- Descending sort by host load ranking (R_H)

4.4 Routing Layer

Experimentation was conducted to determine the viability of available software to modify and use as the Routing Layer.

4.4.1 TCP Hand-off

Initially, one-way architectures were investigated due to their high efficiency and better scalability resulting from outbound packets flowing directly to the client from the pool server rather than through the web switch.

TCP hand-off was the initial choice for the routing layer as the only non-proprietary solution available. Test code from a paper written on the subject [9] was obtained, and an attempt was made to set up the test code. This attempt proved unsuccessful due to the unstable nature of the kernel patch, which was merely for experimental use and was too unstable and "hack" like to be used without significant time being invested. The attempt was abandoned in order to conserve time for the more novel aspects of the project as supposed to reworking an already proven technique.

The decision was made to use a method not requiring kernel modification, described below.

4.4.2 TCP Gateway

This is a two-way architecture and is essentially a standard HTTP proxy server configured to be forward facing. It receives requests from the client, rewrites them and passes them on to a server, reads the response from the server and writes it out to the client.

It is less efficient than TCP hand-off since all packets must flow back through the web switch, and up to the application layer before they are routed, however it allows caching of responses and is the simplest solution to implement since it is entirely at application level.

It was decided to use a TCP Gateway as the routing layer, and evaluate existing Java proxy servers for modification. Due to the separation between the routing layer and the control layer, it remains possible to later modify the system to use TCP hand-off or an alternative routing layer with minimal code changes.

4.4.3 Evaluation of Proxy Servers

Initial Selection

A list of Java proxy servers was identified. Commercial and closed source servers were removed from the list leaving four contenders:

- **JProxy**[13] looked to be designed with the individual user in mind rather than a full-blown proxy server, and did not seem to be particularly advanced (version 01beta)
- **Muffin**[14] is a sourceforge project designed for individual users to use as a content filtering proxy (for removing adverts etc from downloaded pages)
- **RabbIT**[15] is a sourceforge project designed to cache pages, compress pages during transport and also act as a content filtering proxy
- **Scache**[16] is a sourceforge project designed to cache pages and also to allow offline browsing of its cache

Although none of the proxies are designed with a traditional proxy/web cache role in mind (Only the commercial offerings seemed to be designed this way, for example Sun's own Java System Web Proxy Server [17]), it should be possible to easily strip them down to fulfil the requirements of the routing layer.

Evaluation

Jproxy was immediately rejected since it did not seem as suitable as the other three options. The remaining three were then tested in a very quick and rudimentary fashion using autobench [18]. They were also compared to a direct connection to the web server and squid, a C based proxy server which is the industry standard on UNIX based web caches.

The results showed that all proxies performed reasonably well. Eventually Scache was chosen since it had less unwanted features, making the stripping-down process easier.

Initial Implementation

The initial version of the Control Layer was successfully interfaced with Scache despite Scache consisting of somewhat messy code with Czech comments. Some small modifications also had to be made to make the proxy forward-facing, i.e. switching the proxy to handling local requests rather than handling requests to retrieve remote pages.

However once the final version had been developed complete with monitoring data collection, the obfuscated nature of the Scache code provided problems adding timing code. This was because it was difficult to isolate the start and end points of requests since three different functions existed to obtain the page depending on its status in the cache.

A second visit to the cache's homepage to try and obtain newer source revealed that since the first visit, the project had been abandoned by the author and a security notice advised users to stop using Scache due to a remote denial of service vulnerability. Since the steps required to attach the Control Layer to another proxy are minimal, it was decided to abandon modifications to Scache.

Second Implementation

RabbIT was selected as the next choice for modification. The API proved much cleaner and comments were in English. After modification to convert it to a forward-facing proxy, the API was fitted easily. (Full details of how to integrate the API is provided in the documentation accompanying the code)

A bandwidth monitor for the system was created as follows:

rabbit.mercury.OutboundMonitorFilter

Extends java.io.FilterOutputStream. This class is wrapped around all *OutputStream* connections between the web switch and the client, allowing bandwidth usage to be monitored.

`public` **`OutboundMonitorFilter`**`(java.io.OutputStream out)`

Constructs a new *OutboundMonitorFilter* object to wrap the specified *OutputStream*

Method Detail: Constructs the class and registers it by calling the static method *OutboundMonitorThread.AddFilter()*

`public void **write**(byte[] b, int off, int len) throws java.io.IOException`

Writes to the *OutputStream*. Logs the number of bytes written.

`public void **write**(byte[] b) throws java.io.IOException`

Writes to the *OutputStream*. Logs the number of bytes written.

`public void **write**(int b)throws java.io.IOException`

Writes to the *OutputStream*. Logs the number of bytes written.

`public void **close**()`

Closes the *OutputStream*

rabbit.mercury.OutboundMonitorThread

Extends java.lang.Thread, Implements SystemBandwidthMonitor. This class is a background thread that periodically polls total data sent and calculates the current rate. It uses the singleton design pattern (only one thread may be active).

`public static void **SetDelay**(int delay)`

Sets the delay between recalculating bandwidth usage

`public static OutboundMonitorThread **GetMonitor**()`

Returns the currently active thread

`public void **StartMonitor**(mercury.config.MercuryConfig config)`

Starts the thread and loads the configured delay from the configuration store. The new thread calls *setDaemon(true)* in order to ensure it is terminated when the proxy shuts down

`public void **StopMonitor**(mercury.config.MercuryConfig config)`

Stops the thread

`public double **GetCurrentBandwidth**()`

Returns the last calculated bandwidth usage

`public void **run**()`

Loops and periodically calculates the bandwidth usage (period specified by delay)

Method Detail: Initially a static byte counter variable was used for all *OutboundMonitorFilter* objects. This required a synchronized block to prevent updates whilst the data was collected. However because of the potential for this to block all IO from the web switch simultaneously, the model was revised to use a byte counter variable for each object.

This method iterates through all the objects, and calls a synchronized method of each object, which uses a minimal synchronized block to obtain the byte counter value, reset it, and then outside of the synchronized block calculates and returns the rate. All the returned rates are added, and in a synchronized block in this method, the new outbound rate is atomically moved into position.

References to the filter objects are stored using the *java.lang.ref.SoftReference* class which allows garbage collection to destroy the classes if memory is required and they are not referenced by anything other than *OutboundMonitorThread*. Filter objects are also removed from the list if their *close()* method has been called.

If three loops have occurred with no filter objects active, the thread pauses itself by waiting on a lock object. When a new filter is added, the method releases this lock causing the thread to restart.

```
public java.lang.String GetName()
```

Returns "System Outbound Bandwidth Monitor"

4.5 Portability

The system should be developed in Java as previously stated. No native code should be used, providing portability to a large range of platforms.

4.6 Scalability and Resilience

The bottleneck in this system design is the single web switch. In terms of resilience it provides a single point of failure. In terms of scalability it limits the options to a scale-up of the server running the web switch software. As previously discussed there are physical limits to the extent of this scalability.

The system should be designed in such a way that web switches can be pooled in a similar way to the web servers themselves. It should be able to operate in the following configurations:

- **Standalone web switch**: A single web switch, no load-balancing or resilience
- **Failover configuration**: One Active and one or more Hot Standby web switches. The hot standby monitors the active switch and takes over its IP address in the event of a failure. This provides resilience but no load-balancing
- **Load-balancing configuration**: Two or more active web switches share traffic. This is achieved by placing a content blind load balancer in front of the active web switches. Content blind load balancers are able to handle a much greater load because processing occurs at layer 4 of the OSI protocol stack and hence no TCP connection is opened on the load balancer (They simply route traffic) whereas the redirector servers operate at layer 7 of the OSI stack and the user's TCP connection must be terminated at the server.

  This now introduces a new single point of failure in the layer-4 switch, however most implementations of layer-4 switches support failover configurations as described above, and additionally some allow a hardware load-balancing configuration to share traffic between switches.

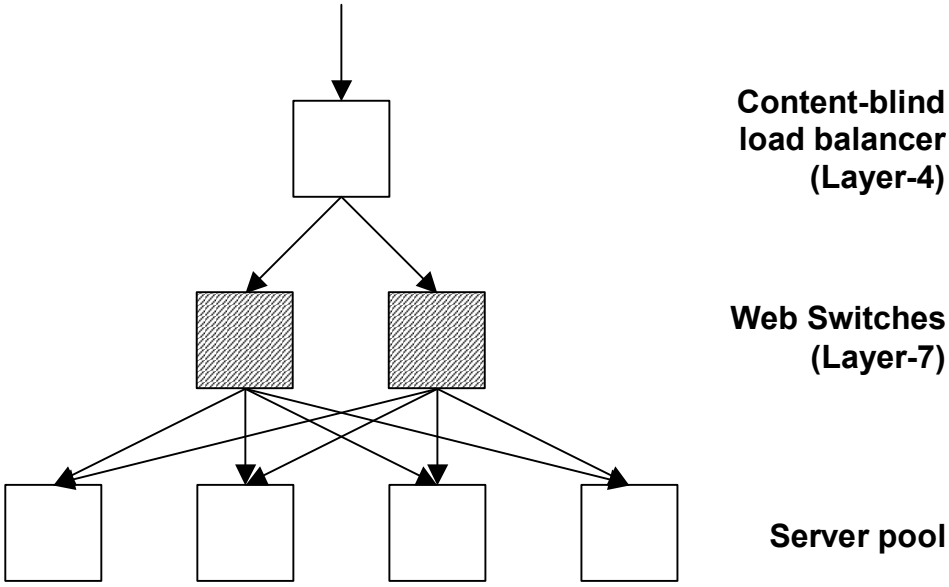

Figure 19 A possible load-balancing configuration of the system

4.7 Summary

The finalised design conforms to the architecture set out in the high level design. It also has the potential to fulfil all of the project goals providing the implementation works as specified in this design section.

Some of the key achievements of this design are:

- Configured by XML file – widely used file format
- VFS config reloads supported atomically without any downtime or inconsistent VFS state, host/path monitoring data is persisted across reloads
- User defined metadata can be added to the XML file and used in plugin classes without any code changes
- Control Layer interface specified using façade design pattern hides internal structure of Control Layer, Interfaces used wherever objects must be returned by functions to further hide internal structure
- Exceptions used to cleanly raise errors outside the Layer
- User plugin classes (Dispatching Algorithm and System Monitor) implement predefined interfaces. Start and stop methods are provided to allow them to load/save configuration. They can be plugged into the system without code changes simply by altering the configuration file
- Normalised host/path statistics stored in time-sliding window array. Averages are cached for speed
- System can be scaled-out if one web switch does not provide suitable capacity
- Connections to backend servers are pooled as a result of the proxy architecture
- HTTP 1.1 and pipelining supported

5 Testing

5.1 Introduction

This chapter details the testing that was undertaken.

Firstly tests were conducted to verify the functionality of the system. Secondly tests were conducted to determine its effectiveness.

5.2 Unit testing

Thanks to the modular design of the system, it was possible to test each part of the system individually after it had been implemented by creating simple test harness classes.

This ensures that each component of the system is bug free and functions as its specification states.

5.3 Integration testing

Once all components of the system were completed, an integration test was conducted by running the application and conducting a number of test scenarios designed to simulate both normal operation and error conditions.

The dispatching algorithm itself was not tested at this point other than to verify that the algorithm's decision was correctly followed by the code (This was achieved by creating an extremely simple algorithm to sort paths in alphabetical order and ensuring the routing layer then tries to access them in this order).

This ensures that the components of the system integrate correctly together and the system as a whole is bug free.

5.4 Effectiveness testing

Once the system had been verified to be bug free, the effectiveness of monitoring data and the dispatching algorithm were evaluated.

5.4.1 Test setup

Apache 1.3.31[23] was installed on a pool of 5 machines. A set of documents was configured on each machine containing simulated HTML pages, CGI Scripts and binary files.

Three versions of the entire virtual server file tree were provided on each machine, a high, medium and low bandwidth version, at /HI (Containing large files), /MED (Containing medium files) and /LOW (Containing small files) respectively.

Two CGI scripts were provided, MED.cgi which was a very simple Perl CGI script, and HI.cgi which was a Perl script which also contains a 1 million iteration loop inside which the value of a variable is changed several times. This should ensure that the scripts are affected proportionately by load (Since the second one should be somewhat more IO bound, although the single variable will almost certainly be cached by the OS). The *target* response time average/standard deviation was calculated under a light load. The base virtual file system configuration for all tests was the following:

```
<servermanifest>
<virtualfile path="/html/">
<file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync20:7962/HI/html/"/>
<file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync21:7962/HI/html/"/>
<file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync22:7962/HI/html/"/>
<file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync23:7962/HI/html/"/>
<file bandwidth="1" load="1" average="6" stdev="83"
url="http://sync24:7962/HI/html/"/>

<file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync20:7962/MED/html/"/>
<file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync21:7962/MED/html/"/>
<file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync22:7962/MED/html/"/>
<file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync23:7962/MED/html/"/>
<file bandwidth="0.5" load="1" average="6" stdev="83"
url="http://sync24:7962/MED/html/"/>

<file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync20:7962/LOW/html/"/>
      <file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync21:7962/LOW/html/"/>
<file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync22:7962/LOW/html/"/>
```

```
<file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync23:7962/LOW/html/"/>
<file bandwidth="0" load="1" average="6" stdev="83"
url="http://sync24:7962/LOW/html/"/>
</virtualfile>
<virtualfile path="/cgi-bin/test.cgi">
<file load="1" bandwidth="1" average="2830" stdev="2825"
url="http://sync20:7962/cgi-bin/HI.cgi"/>
<file load="1" bandwidth="1" average="2830" stdev="2825"
url="http://sync21:7962/cgi-bin/HI.cgi"/>
<file load="1" bandwidth="1" average="2830" stdev="2825"
url="http://sync22:7962/cgi-bin/HI.cgi"/>
<file load="1" bandwidth="1" average="2830" stdev="2825"
url="http://sync23:7962/cgi-bin/HI.cgi"/>

<file load="1" bandwidth="1" average="2830" stdev="2825"
url="http://sync24:7962/cgi-bin/HI.cgi"/>
<file load="0.5" bandwidth="1" average="131" stdev="1088"
url="http://sync20:7962/cgi-bin/MED.cgi"/>
<file load="0.5" bandwidth="1" average="131" stdev="1088"
url="http://sync21:7962/cgi-bin/MED.cgi"/>
<file load="0.5" bandwidth="1" average="131" stdev="1088"
url="http://sync22:7962/cgi-bin/MED.cgi"/>
<file load="0.5" bandwidth="1" average="131" stdev="1088"
url="http://sync23:7962/cgi-bin/MED.cgi"/>
<file load="0.5" bandwidth="1" average="131" stdev="1088"
url="http://sync24:7962/cgi-bin/MED.cgi"/>

<file load="0" bandwidth="1" average="14" stdev="865"
url="http://sync20:7962/HI/html/1297.html"/>
<file load="0" bandwidth="1" average="14" stdev="865"
url="http://sync21:7962/HI/html/1297.html"/>
<file load="0" bandwidth="1" average="14" stdev="865"
url="http://sync22:7962/HI/html/1297.html"/>
<file load="0" bandwidth="1" average="14" stdev="865"
url="http://sync23:7962/HI/html/1297.html"/>
<file load="0" bandwidth="1" average="14" stdev="865"
url="http://sync24:7962/HI/html/1297.html"/>
</virtualfile>
</servermanifest>
```

During all tests, the caching functionality of the routing layer (RabbIT) was completely disabled. The access logging system of the routing layer was configured to show the request virtual path, and the physical path that it was mapped to. The *SystemTrace* class was used to show a GUI window refreshing every second giving full details of the internal state of all components of the Control Layer.

Siege[24] was used to generate load, running on the web switch in order to eliminate network delay. Siege was used because it allows the maximum number of concurrent connections to be limited, so load can be tested incrementally.

Test results were obtained by creating a simple test harness class, which starts the proxy, runs siege and allows a few seconds for the load to stabilise, then runs the desired tests. Results are dumped out to CSV format for import to Microsoft Excel.

5.4.2 List of tests conducted

I. System Bandwidth monitor

Purpose

Ensure the system bandwidth monitor is reporting correct values.

Dispatching Algorithm

Static algorithm configured to sort physical paths alphabetically.

Setup

Siege is configured to request a single URL for a duration of 5 minutes. In each test run, the reported value of outbound bandwidth is recorded every minute and an average is taken. The outbound bandwidth is allowed to drop back to 0 before commencing each test run.

Variables

Maximum number of concurrent connections is increased in each experiment from 5-50 connections using a step size of 5.

Expected Result

As maximum number of concurrent connections increases, the total bandwidth usage should increase proportionally until a limiting factor such as the web switch's maximum number of threads is reached.

Result

The following graph was obtained:

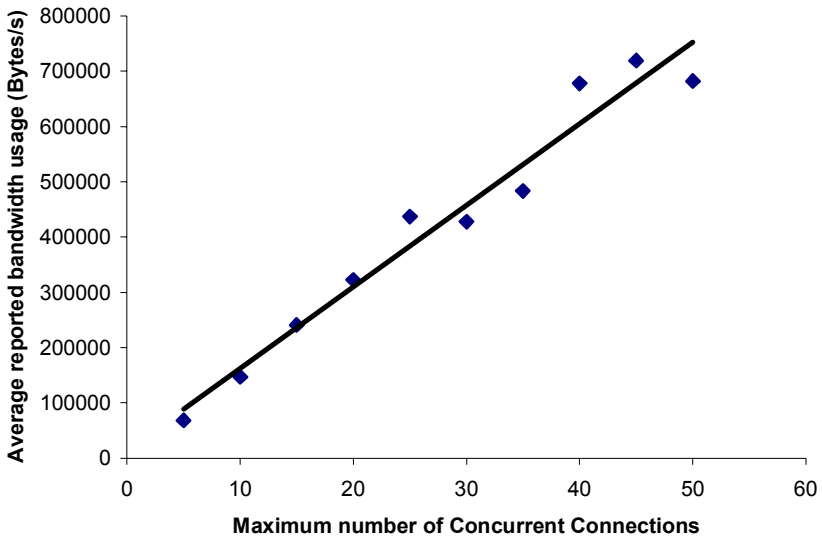

Figure 20 Average reported bandwidth usage compared to number of concurrent connections (Line shows linear trendline)

Conclusion

As expected, reported bandwidth usage is directly proportional to maximum number of concurrent connections.

II. Host connection counter

Purpose

Ensure the host connection counter is reporting correct values.

Dispatching Algorithm

Static algorithm configured to sort physical paths alphabetically.

Setup

Siege is configured to request a single URL for a duration of 5 minutes. In each test run, the reported number of connections to the host is recorded every minute and an average is taken. The connection count is allowed to drop back to 0 before commencing each test run.

Variables

Maximum number of concurrent connections is increased in each experiment from 5-50 connections using a step size of 5.

Expected Result

As maximum number of concurrent connections increases, one host's connection counter should increase proportionally. It will not be exactly equal due to the pooling of spare connections and additional delays in tearing down connections after the client has closed them. These factors will also cause some error. Note

that only one host's connection counter will change because the dispatching algorithm will send all requests to the same host.

Result

The following graph was obtained:

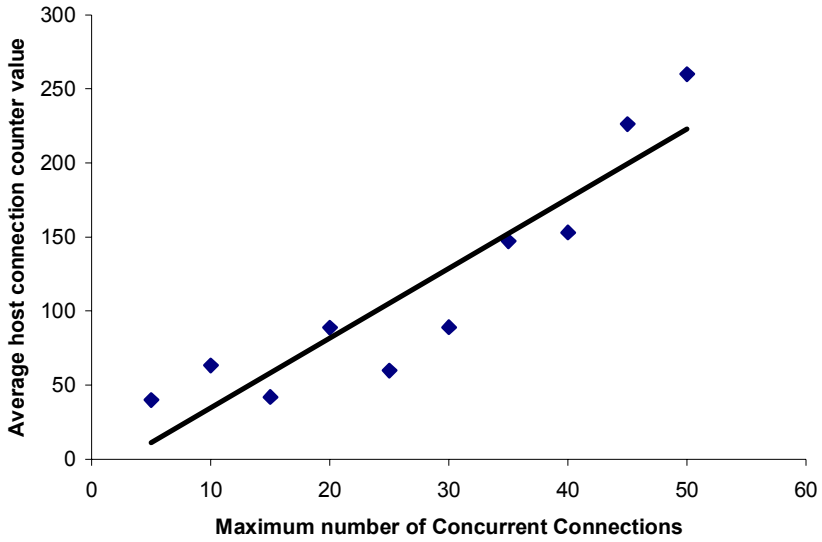

Figure 21 Average host connection counter value compared to number of concurrent connections (Line shows linear trendline)

Conclusion

As expected, the host connection counter value is directly proportional to maximum number of concurrent connections, allowing for some deviation as described above.

III. Bandwidth Throttle Value

Purpose

Ensure the internal bandwidth throttle is set correctly.

Dispatching Algorithm

Static algorithm configured to sort physical paths alphabetically, bandwidth throttle calculation enabled as described in section 4.3.8.

Setup

The maximum outgoing bandwidth is set at 500,000 bytes/second. According to test I this value should be reached at around 35 concurrent connections.

Siege is configured to request a single URL for a duration of 5 minutes. In each test run, the reported value of the bandwidth throttle is recorded every minute

and an average is taken. The throttle value is allowed to return to 1 before commencing each test run.

Variables

Maximum number of concurrent connections is increased in each experiment from 5-50 connections using a step size of 5. Between 20-35 connections a reduced step size of 1 was used.

Expected Result

The throttle should remain at 1 until 80% of the maximum bandwidth is reached. It should then begin to drop to 0 and reach 0 when 100% of the maximum bandwidth is reached. Note that the dispatching algorithm is static, so the system will not try to compensate for the bandwidth usage. The graph obtained should be similar to Figure 17 in section 4.3.8.

Result

The following graph was obtained:

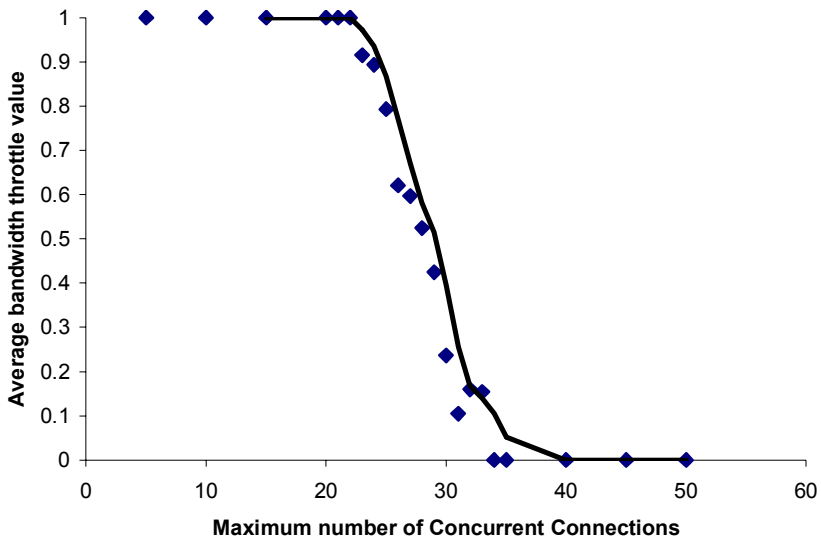

Figure 22 Average bandwidth throttle value compared to maximum number of concurrent connections (Line shows moving average, period 3)

Conclusion

As expected, the bandwidth throttle value fits the expected curve with some deviation. This deviation is because the relationship between measured outbound bandwidth and maximum number of concurrent connections is a dynamic equilibrium, so some the relationship between bandwidth throttle value and concurrent connections also becomes a dynamic equilibrium.

IV. Bandwidth Throttle Operation

Purpose

Ensure the system delivers alternative versions of content correctly according to the bandwidth throttle value

Dispatching Algorithm

Configured to sort only on path bandwidth ranking (described in section 4.3.8). Paths with an equal ranking value are sorted alphabetically.

Setup

The maximum outgoing bandwidth is set at 500,000 bytes/second. According to test I this value should be reached at around 35 concurrent connections.

Siege is configured to request a single URL for a duration of 5 minutes. In each test run, the proportion requests receiving each type of page (HI/MED/LOW) is recorded. The throttle value is allowed to return to 1 before commencing each test run.

Variables

Maximum number of concurrent connections is increased in each experiment from 5-50 connections using a step size of 5. Between 20-35 connections a reduced step size of 1 was used.

Expected Result

At low connection values 100% of requests should receive HIGH quality pages. At round 25 connections this should tail off. At high connection values 100% of requests should receive LOW quality pages. In the middle MED pages should peak.

However because of the time taken for the throttle value to dropping from 1, it is likely that HIGH quality pages will always account for a certain proportion of responses. Also it is unlikely that the system will establish a completely stable state since it operates as a dynamic equilibrium.

Result

The following graph was obtained:

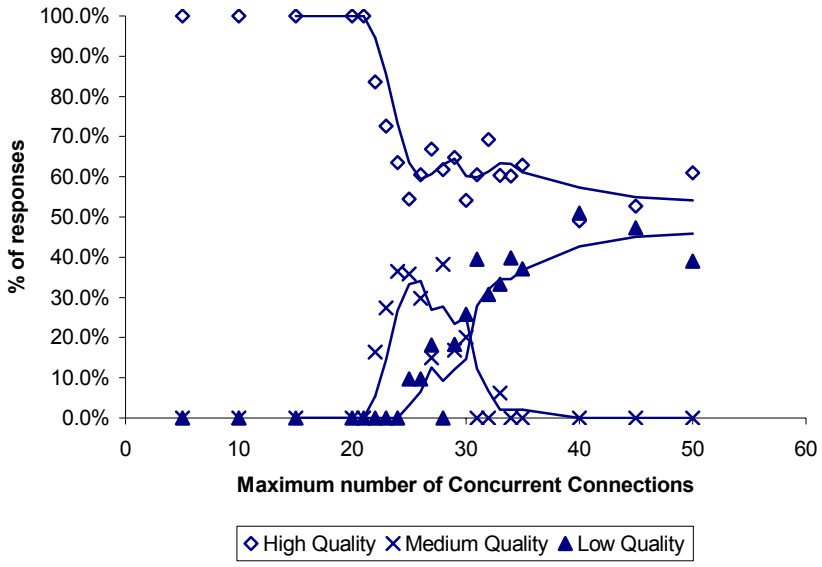

Figure 23 Response types compared to maximum number of concurrent connections (Lines show moving averages, period 3)

Conclusion

As expected, 100% of responses are high quality at low numbers of concurrent connections, and medium quality responses peak in the centre of the graph. However, at higher numbers of concurrent connections the low/high quality responses seem to level off.

The reason this occurs is as follows: Initially, the bandwidth throttle starts at 1 and high quality responses are sent, causing bandwidth usage to soar. The bandwidth usage is updated every 5 seconds and once the first update has occurred, the throttle drops to 0 due to the huge bandwidth usage. This causes low quality responses to be sent, which causes the cycle to repeat. Hence the bandwidth throttle value oscillates between 0 and 1, causing the responses sent to be roughly equal in proportion.

V. Redesigned Bandwidth Throttle

Modifications

The formula for calculating the system bandwidth throttle was modified to include damping:

$$T_B = \left(0.1 \times \sqrt{\min\left(\max\left(\frac{\left(\frac{R_{current}}{R_{\max}} - 0.8 \right)}{0.2}, 0 \right), 1 \right)} \right) + \left(0.9 \times LastT_B \right)$$

The system bandwidth throttle was also prevented from being changed more than once per second. This ensures that it can only alter by 0.1 each second. Initially this damping was found to cause the throttle to never quite reach 1, preventing the highest bandwidth pages from being sent. This was rectified by rounding the value to 3 d.p.

Additionally the update speed of the system bandwidth monitor was increased to 1 second. This ensures that the results of changes in the throttle value are propagated back quickly.

As a result of this increased update frequency, it was discovered that the *OutboundMonitorThread* class was not thread safe – its internal list of registered monitors was being added to by new connections whilst the thread was using an *Iterator* to calculate the bandwidth used, causing unpredictable behaviour and incorrect bandwidth usage to be reported. *OutboundMonitorThread* was modified to place newly registered filters onto a queue and then have them added to the main list of registered filters by the *Thread* itself. This reduces the amount of time spent in *synchronized* blocks, increasing efficiency.

Retesting

Since the bandwidth monitor had been updated, test I was repeated this time using 10 averages per point. The following graph was obtained (Standard deviation of the 10 averages for each result was calculated and used to add Y error bars of length 1 standard deviation):

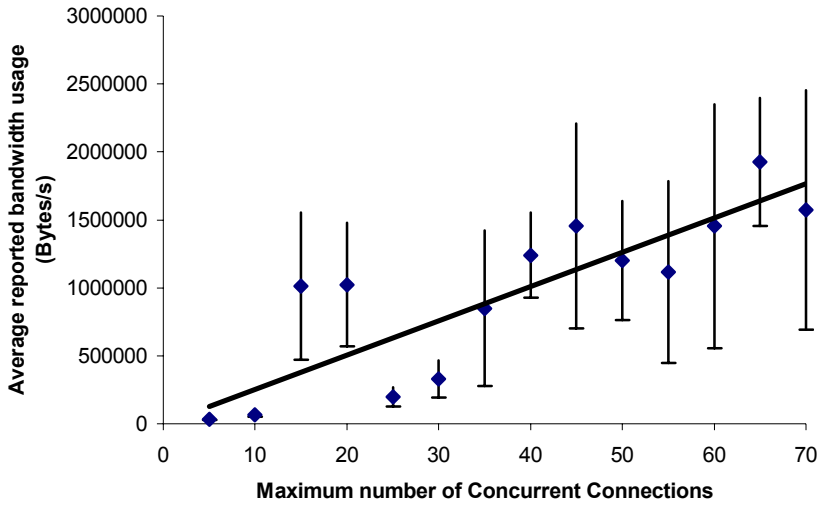

Figure 24 Average reported bandwidth usage compared to number of concurrent connections (Line shows linear trendline)

Test II was not repeated because the modified bandwidth monitor does not affect the connection counter.

Test III was repeated and error bars were added in the same way as Figure 24:

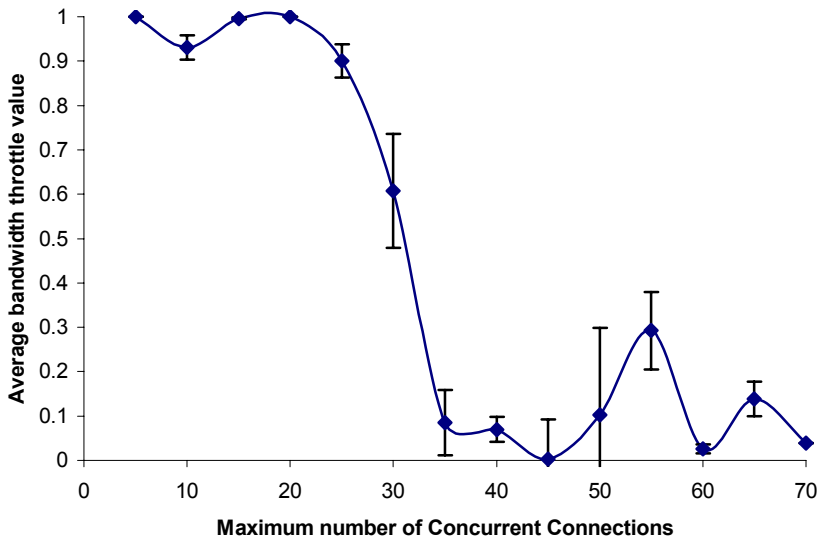

Figure 25 Average bandwidth throttle value compared to maximum number of concurrent connections

Test IV was also repeated, giving the following graph:

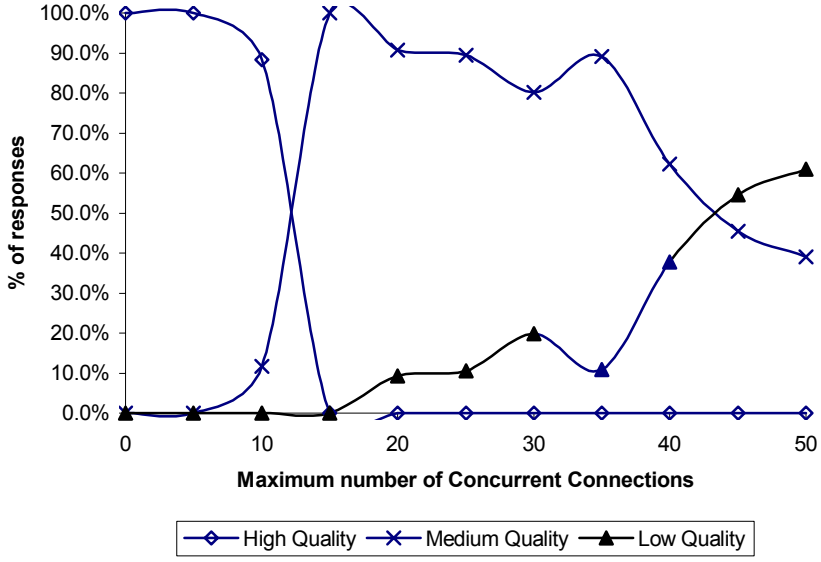

Figure 26 Response types compared to maximum number of concurrent connections

Conclusion

Figure 24 shows reported bandwidth usage remains proportional to concurrent connections, although the error is greater. This is because the bandwidth usage is being calculated over a 1 second period rather than a 5 second period, so the data is much more susceptible to the bursty nature of network traffic. The error bars show there is indeed a reasonable fit, well within 1 standard deviation in most cases.

Figure 25 gives a similar shaped graph to the previous result, however the error is much greater. This is most likely due to the

Figure 26 shows that the redesigned algorithm is producing a graph which looks exactly like the predicted shape in section IV. The other graphs show that the redesigned algorithm has not significantly changed anything else.

VI. Host Connection Load Balancing

Purpose

Ensure the host connection load balancing algorithm operates correctly

Dispatching Algorithm

Configured to sort only on host load ranking (described in section 4.3.8). Paths with an equal ranking value are sorted alphabetically.

Setup

Siege is configured to request a single URL for a duration of 10 minutes. In each test run, the reported number of active connections to each host is recorded every minute and an average is taken.

Variables

Maximum number of concurrent connections is increased in each experiment from 5-45 connections using a step size of 10.

Expected Result

All hosts should have an equal connection load.

Result

The following graph was obtained:

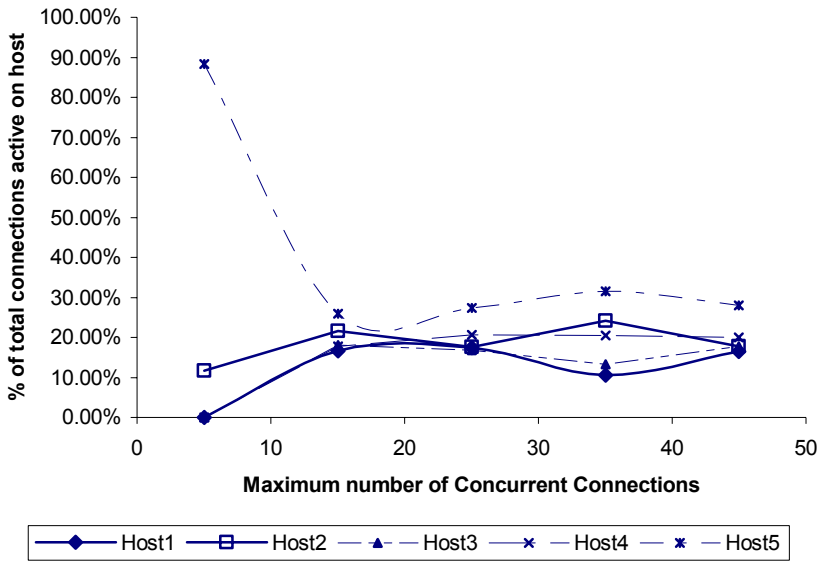

Figure 27 Average percentage of connections on host compared to maximum number of concurrent connections

Conclusion

Excepting one outlying point under low connection load (When the system is least stable because the low volume of traffic tends to allow oscillations to occur more easily), the connection load seems to be stable across hosts as predicted.

VII. Host Load Balancing

Purpose

Ensure the host load balancing algorithm operates correctly

Dispatching Algorithm

Configured to sort on host *average standardised measured response time* (described in section 4.3.8) followed by host load ranking. Paths with an equal ranking value are sorted reverse alphabetically so that the MED cgi script is used in all cases. This ensures that the host load affects the response times.

Setup

Siege is configured to request a single URL with a maximum of 30 concurrent connections. At the end of each run the number of connections made to each host are counted. During experimentation two hosts are heavily loaded using stress[25]

Variables

Two hosts are heavily loaded, the other 3 are not loaded. 3 tests were run, lasting 1 minute, 30 seconds and 15 seconds.

Expected Result

Requests should be shared equally between the 3 unloaded hosts, the 2 loaded hosts should receive less requests.

Result

The following graph was obtained:

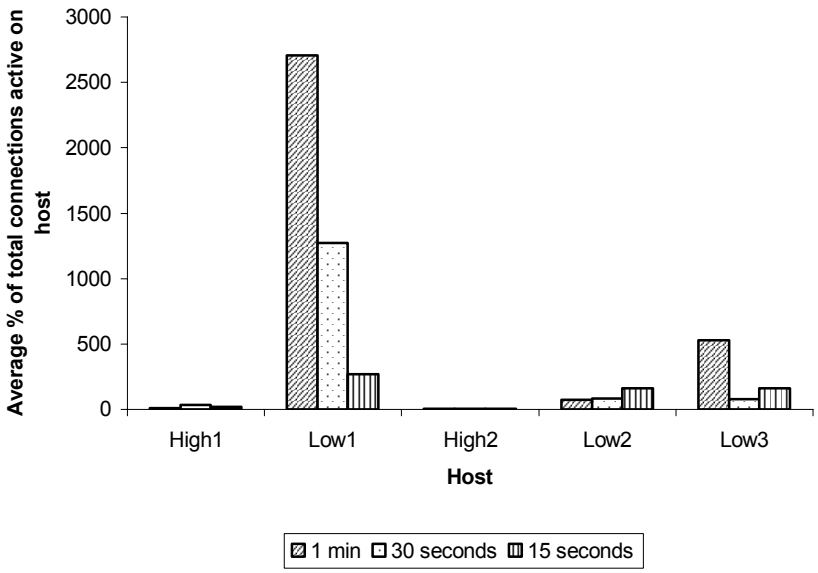

Figure 28 Average percentage of connections per host

Conclusion

It can be seen that the dispatching algorithm quickly learns not to send requests to the highly loaded hosts in all cases. However on longer runs the system builds up a prejudice towards one host. This is because the low loaded hosts do not

respond in exactly equal time, so a prejudice develops. Once this has happened, the majority of requests go to this host, and eventually no requests go to the other hosts. This causes their monitoring data to remain negative and stagnant, so no requests will be sent to them until the monitoring data expires. Further investigation is necessary to determine how to deal with this situation.

VIII. Path Load Balancing

Purpose

Ensure the path load ranking algorithm operates correctly

Dispatching Algorithm

Configured to sort on path load ranking (described in section 4.3.8).

Setup

Only one pool server was used for this test. Siege is configured to request a single URL for a duration of 5 minutes. In each test run, the proportion requests receiving each type of page (HI/MED/LOW load) is recorded. The throttle value is allowed to return to 1 before commencing each test run.

Variables

Maximum number of concurrent connections is increased in each experiment from 5-50 connections using a step size of 5.

Expected Result

The expected result is the same as that of test IV. This is assuming the different versions of the content respond proportionally worse under system load (I.e. if system load is increased, high, medium and low response times are increased by a factor of H, M and F respectively where H > M > F).

Result

The following graph was obtained:

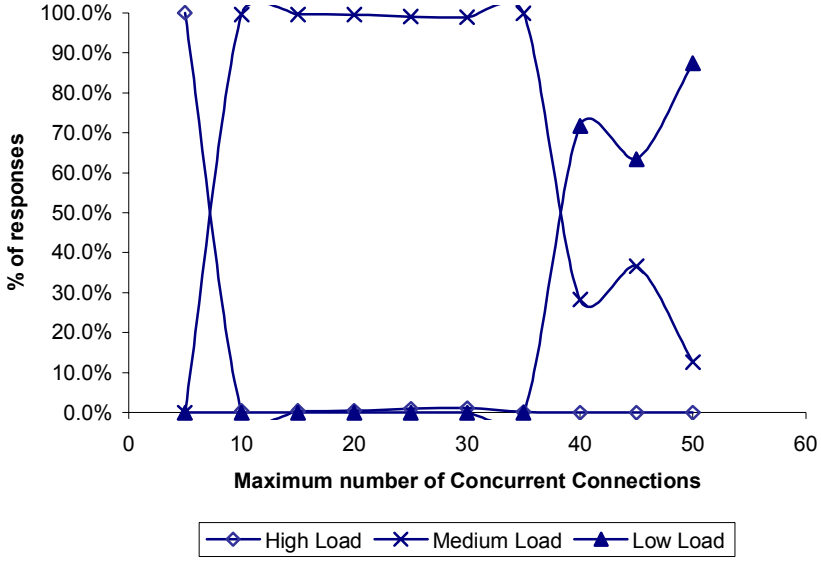

Figure 29 Response types compared to maximum number of concurrent connections

Conclusion

The resulting graph is exactly as expected.

IX. Overall performance

Purpose

To verify the advantage of using this system

Dispatching Algorithm

The complete algorithm (described in section 4.3.8).

Setup

Siege is configured to request a range of URLs on the server for a duration of 5 minutes. In each test run the throughput and average response time is recorded.

Variables

The test is run on the test VFS system. It is then re-run but with only the HIGH load/bandwidth versions of content present.

Expected Result

The expected result is that due to the system's ability to throttle connections, a higher throughput and lower response time can be sustained with the alternative versions of content in place.

Result

The following graphs were obtained:

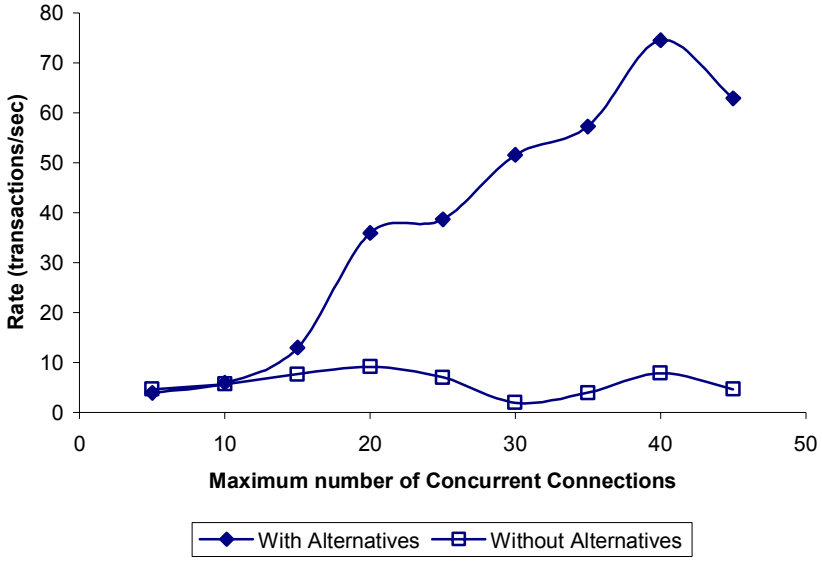

Figure 30 Transaction rate against maximum connections with/without alternative page versions

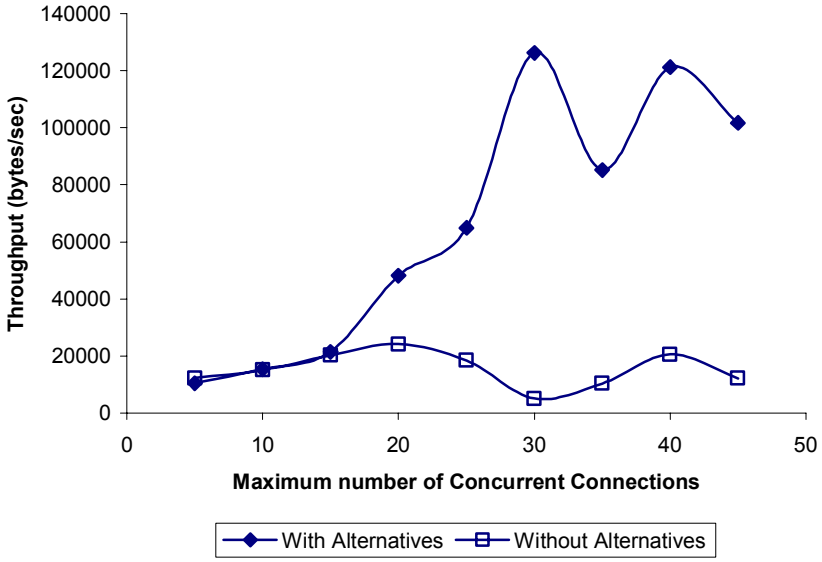

Figure 31 Throughput against maximum connections with/without alternative page versions

Conclusion

The graphs show that the throttling algorithm provides a clear increase in scalability compared to the same system running with the throttling algorithm effectively disabled.

6 Evaluation

6.1 Introduction

This section evaluates the project in terms of its goals, and attempts to evaluate how well each goal has been fulfilled.

6.2 Summary of Goals

In section 1.2 the goals of the project were defined to be producing a system which fulfils the following criteria:

- **Throttling**
- **Load balancing**
- **Heterogeneous pool**
- **Adaptable**

As shown by experimental evidence in the previous chapter, the system correctly throttles and load balances connections (With the exception of the limitations detailed below). It also provides an improvement over the same system without throttling (Note the system was not compared to other systems because the implementation of this routing layer is designed to be a proof of concept rather than optimal)

By design the system supports a heterogeneous pool of servers, this was verified in integration testing.

Similarly by design the system is adaptable. Firstly various parameters are configurable, for example the weights on the default load balancing algorithm can be changed to alter the system's behaviour. If the user desires to make major alterations to the system's behaviour, user defined system monitors and load balancing algorithms can be implemented and plugged into the system with no code changes to the system itself. Additionally the system can be retro-fitted to any layer-7 web switch that is able to interface with Java.

The statistics array classes introduces some problems due to it causing feedback – it affects whether or not a host/path is accessed, and once its values cause the host/path to stop being accessed until the values in the array time out, a

discussion of possible further work to overcome these is discussed in the next section.

7 Conclusion

7.1.1 Limitations

This section identifies the limitations of the system and highlights possible further work.

The problems mentioned in the previous section caused by the statistics array could be solved by damping the average value to ensure the system's state changes more steadily.

Alternatively, a possible extension to the project would be to investigate the possible use of Change Detection Algorithms[26]. These do not require prior knowledge of the typical performance of the system they are monitoring, and instead build this up as they run. When a statistically significant change occurs it is detected.

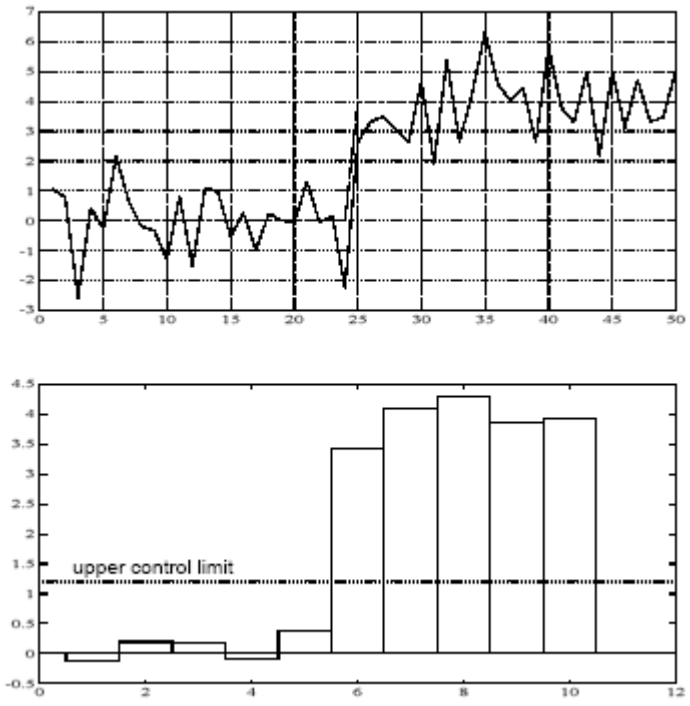

Figure 32 A Shewhart control chart (below) corresponding to a change in the mean of a Gaussian sequence with constant variance [26]

Another problem would be caused by a factor such as backend database load was causing a slowdown – if one pool server contained more scripts relying on the database than others, the server would be unfairly penalised since it's average response times would be higher than a server with a higher proportion of static pages. Extension work could look into ways of detecting and compensating for this.

Finally, if a proxy server was to be placed between the web switch and server pool it would cause error in the results due to caching, it is difficult to know whether to use a No-Cache: header to get accurate results or not use the header in order to gain benefit from caching

7.1.2 Extensions

Possible extensions to the project include:

- If 2 or more web switches are being used in a load balancing configuration, monitoring data could be shared via the network
- An algorithm could be added to periodically test infrequently accessed pages to maintain monitoring data
- The algorithms used in the system could be formally proved using a network simulator[27]
- Further investigation of the system's behaviour over very low bandwidth connections is required to determine the effectiveness of the system bandwidth monitors; in particular these need to be evaluated to ensure that the system's buffering (in the case of outbound monitors) or lack thereof (in the case of monitoring pool servers where the incoming connection may be waiting on the client receiving data) does not cause incorrect results

7.1.3 Summary of achievements

A system has been constructed which fulfils the initial goals. The system builds upon existing research in this field, and provides a modular framework within which users can specify load balancing algorithms. These algorithms are presented with a base set of metadata and monitoring data which users can add to without making code changes to existing classes. Testing has proved that these monitors yield the expected values according to the state of the system.

The load balancing algorithm investigated as the system default introduces the novel concept of throttling. Testing has proved that this can provide increased throughput compared to systems without this

The system has been implemented in a scalable and portable manner, enabling it to be used on a variety of platforms on web sites of all popularities and sizes.

8 Bibliography

[1] V. Cardellini, E. Casalicchio, M. Colajanni, P. Yu. The State of the Art in Locally Distributed Web-Server Systems, *ACM Computing Surveys*, Vol. 34, No. 2, June 2002, pp. 263-311

[2] http://news.netcraft.com/archives/web_server_survey.html

[3] http://hotwired.lycos.com/webmonkey/templates/print_template.htmlt?meta=/webmonkey/geektalk/97/12/index4a_meta.html

[4] http://homepage.ntlworld.com/martin.hubert1/osi.htm

[5] http://java.sun.com/j2se/1.4.2/

[6] http://www.eclipse.org/

[7] http://www.gnu.org/software/cvs/cvs.html

[8] http://wwws.sun.com/software/sundev/whitepapers/java-style.pdf

[9] R. Kokku, R. Rajamony, L. Alvisi, H. Vin. *Half–pipe Anchoring: An Efficient Technique for Multiple Connection Handoff*, 2002

[10] M. Mitzenmacher. How useful is old information, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, January 2000, pp. 6–20

[11] http://www.w3.org/CGI/

[12] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel. *Locality-Aware Request Distribution in Cluster-based Network Servers*, (1998)

[13] http://www.geocities.com/SiliconValley/Bay/8925/jproxy.html

[14] http://muffin.doit.org/

[15] http://rabbit-proxy.sourceforge.net

[16] http://scache.sourceforge.net/

[17] http://wwws.sun.com/software/products/web_proxy/home_web_proxy.html

[18] http://www.xenoclast.org/autobench/

[19] E. Bletsas, *Aeolus: A Web Server Benchmark for Dynamic Content*, 2003

[20] J. Jung, B. Krishnamurthy, M. Rabinovich. *Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites*, 2002

[21] http://www.jdom.org/

[22] X. Chen, J. Heidemann, *Flash Crowd Mitigation via Adaptive Admission Control based on Application-level Observations*, 2002

[23] http://www.apache.org/

[24] http://www.joedog.org/siege/

[25] http://weather.ou.edu/~apw/projects/stress/stress.html

[26] http://www.irisa.fr/sisthem/kniga/file2.pdf

[27] http://www.isi.edu/nsnam/ns/

[28] http://sax.sourceforge.net/