

UNIVERSITY OF OSLO
Department of Informatics

Performance
Evaluation of the
Apache Traffic Server
and Varnish Reverse
Proxies

Shahab Bakhtiyari
Network and System Administration
University of Oslo

May 23, 2012



Performance Evaluation of the Apache Traffic Server and Varnish Reverse Proxies

Shahab Bakhtiyari
Network and System Administration
University of Oslo

May 23, 2012

Contents

1	Introduction	8
1.1	Why Cache?	8
1.2	Motivation	9
1.3	Problem statement	11
1.4	Thesis Outline	11
2	Background and Related Workj	12
2.1	Web servers	12
2.1.1	Static web resources	12
2.1.2	Dynamic web resources	13
2.2	Cache servers	13
2.2.1	Client side proxy	14
2.2.2	Organization and ISP proxy caches	14
2.2.3	Server ISP or CDN reverse proxy caches	14
2.2.4	Server side reverse proxy cache	15
2.2.5	Distributed Caches and ICP	15
2.3	Cache replacement algorithms	16
2.3.1	Replacement strategies	16
2.4	HTTP	21
2.4.1	HTTP Message Structure	22
2.5	Caching software	25
2.5.1	Apache Traffic Server	25
2.5.2	Varnish	27
2.5.3	Others	28
2.6	Challenges	28
2.6.1	Realistic Workloads	28
2.6.2	Lack of recent works	30
2.6.3	Tools specifically designed for cache benchmarking	30
3	Model and Methodology	32
3.1	Approach	32
3.2	Test environment	34
3.3	Web Polygraph	36
3.3.1	Polygraph testing phases	37
3.3.2	Command line options	39
3.3.3	Polygraph console output	40
3.4	Surrogate tests	41

3.4.1	Blktrace	41
3.4.2	Seekwatcher	42
3.5	Varnish Configuration	43
3.6	Traffic Server Configuration	44
3.7	The custom scripts	44
3.8	Polygraph Configuration	46
3.8.1	IP Addresses	46
3.8.2	Working Set Size	47
3.8.3	Cache size	48
3.8.4	Content types	49
3.8.5	The phases	50
3.8.6	Server configuration	51
3.8.7	Client configuration	51
3.9	Defining the Experimental Phases and Workloads	52
3.9.1	The best effort method	52
3.9.2	Baseline Workloads	53
3.9.3	The increasing rate workload	54
4	Results and Analysis	55
4.1	Sample Polygraph results	55
4.2	Best effort workload results	57
4.3	Single content type workload results	60
4.3.1	Results for image content type	60
4.3.2	HTML content type results	64
4.3.3	Download content type results	67
4.3.4	Summary and Discussion	70
4.4	Results for the mixed content, increasing rate experiments	72
4.4.1	Results for the top1 phase	73
4.4.2	Increasing traffic rate phase inc	76
4.4.3	Summary and Discussion	80
4.5	Surrogate data results	82
5	Discussion	88
5.1	Summary of the results	88
5.2	Analysis of the ATS saturation point	89
5.3	Pitfalls and Issues with the Server Software	91
5.4	Difficulties with the Polygraph tool	92
5.5	Open Issues and Future Work	93
6	Conclusion	94
6.1	Appendix: The automation and Surrogate scripts	99
6.2	Appendix: The baseline, configuration files	105

List of Figures

2.1	Apache TrafficServer processes	26
2.2	Popularity of web objects follows the Zipf-like Law with α close to 1. Figures illustrate a Zips-Like law distribution with $\alpha=0.90$	29
2.3	Zipf versus uniform popularity models, taken from <i>Cacheoff</i> [1]	30
3.1	Different possible experimental approaches	34
3.2	Experimental private network setup	35
3.3	Phases	38
3.4	The current phase state: client side console output	40
3.5	The runtime output: client side console	41
3.6	Content type distribution	53
4.1	The General form of Polygraph result plots	56
4.2	The of Polygraph result scatter plots	56
4.3	Best effort request rate and throughput	57
4.4	Best effort response times	58
4.5	Best effort document and byte hit rates	58
4.6	Best effort response time vs request rate distribution, client side	59
4.7	Best effort response time vs request rate distribution, server side	59
4.8	Image content type throughput	61
4.9	Image content type response time trace	62
4.10	Image content type document and byte hit rates	63
4.11	Image content type response rate vs response time	63
4.12	HTML content type throughput	64
4.13	Response time trace, Varnish	65
4.14	Response time trace, ATS	65
4.15	HTML content type document and byte hit rates	66
4.16	HTML content type response time vs response rate, client side	66
4.17	HTML content type response time vs response rate, server side	67
4.18	Throughout results for the Download content type	68
4.19	Response times for Download content type	68
4.20	Download content type document and byte hit rates	69
4.21	Download content type response time vs response rate	69
4.22	Summary of the Baseline tests	70
4.23	Mixed content workload content type distribution	73
4.24	Modest request rate throughput	73
4.25	Response time results for top1 phase	74
4.26	Document and byte hit rates, top1 phase	75

4.27	Response time vs response rate, <i>top1</i> phase	76
4.28	Mixed content workload phase <i>inc</i>	77
4.29	Varnish response time, <i>inc</i> phase	78
4.30	ATS response time, <i>inc</i> phase	78
4.31	Hit rates during the <i>inc</i> phase	79
4.32	Response time vs response rate, <i>inc</i> phase	80
4.33	Performance during the constant rate phase	80
4.34	Performance during the increasing rate phase	81
4.35	CPU and Memory Usage, Best Effort Workload	82
4.36	Disk I/O - mixed content type, increasing rate	83
4.37	CPU/memory usage, single content type workloads	84
4.38	Disk I/O - HTML content type	85
4.39	Disk I/O for Image content type	85
4.40	Disk I/O - Download content type	85
4.41	CPU/memory usage, mixed content type workload	86
4.42	Disk I/O, mixed content type workload	87
5.1	Concurrent HTTP connections for ATS	89
5.2	Concurrent HTTP/TCP connections for ATS	90
5.3	Concurrent HTTP connections for Varnish	90
5.4	Concurrent HTTP/TCP connections for Varnish	91

List of Tables

2.1	The Cache Control directives	23
2.2	HTTP protocol methods	25
2.3	Replacement policies for RAM and Disk, Varnish vs ATS	28
3.1	Test hardware specifications	35
3.2	IP addresses	36
3.3	Parameters for the Best Effort workload	53
3.4	The Phases for the Best Effort Workload	53
3.5	The Phases for each individual content type workload	54
4.1	Image workload defined and generated parameters	60
4.2	HTML workload defined and generated parameters	64
4.3	Download workload defined and generated parameters	67
4.4	The better performer of each scenario	71
4.5	The workload phases for the mixed content type experiment	72
4.6	Mixed content workload parameters	72
4.7	Summary of results from the <i>inc</i> phase	82
4.8	Mean %CPU usage for various workloads (2 cpus)	83

LIST OF TABLES

â&O

Abstract

The aim of this thesis was to investigate different performance aspects of two reverse proxy cache servers which are called Varnish and Apache Traffic Server. It uses the tool Web Polygraph to generate various types of web traffic workloads. Both artificial and realistic workloads were designed and generated for each proxy in an identical test set up. In addition several system metrics (so-called Surrogate tests) were collected simultaneously to have the overview of overall system performance. For the experiments conducted in this research, the results indicated that Apache Traffic Server reached better cache hit rates and slightly better bandwidth throughput with the cost of higher system and network resource usage. Varnish on the other hand managed to response higher request rates with better response time, especially for the cache hits. The findings in this thesis indicates that Varnish seems to be more promising reverse proxy.

Acknowledgement

I would like to use this opportunity to thank all of people who helped me along with this work.

My special thanks goes to my wonderful supervisor AEleen Frisch who dedicated a lot of her time, always being available for me, providing me needed resources, ideas, encouragement when things didn't go well. Without her support thing would have been more difficult.

Secondly I want to thank University of Oslo and Oslo University collage ,with all my teachers during this master program Especially Haarek Haugerud,Kyrre Begnum and Ismail Hassan who provided us a friendly and instructive environment during this master program.

I want to thank my beloved family and specially sister whose sacrifices allowed me to be here and finish this work.

Last but not least, I thank all of my friends and classmates whose inspiring discussions and tips helped me through doing my thesis.

Chapter 1

Introduction

In early days of the Internet, when there were only a small number of web objects in existence, which hardly exceeded a couple of hundred thousand, one could easily search for any web resource and retrieve it from the origin server relatively fast due to the ease of indexing. Today, however, each search engine indexes billions of objects, which are only a fraction of all of the materials found on the Internet[2]. From the user's point of view, the time required to access a web page matters. The "eight-second rule" states that when there is a significant likelihood of losing a website visitor if the request is not satisfied within 8 seconds. Users quickly move to another website if they are not happy with the current website's response[2]. Caching was originally introduced to reduce the response time for a request experienced by the client.

The enormous and rapid growth of the Internet in recent years continues, with the number of users continually getting larger. Users nowadays use multiple devices which are connected to internet. One estimation says in 2012 the number of mobile devices connected to the Internet exceeds the whole world's population, and the global IP traffic will increase fourfold by 2015 when the annual IP traffic will reach the zeta byte (10^{21} byte) threshold. The number of devices connected to IP networks will be twice as high as world's population in 2015.[3].

1.1 Why Cache?

Caches are intermediary servers which shorten the path between clients and servers by storing web resources. Caching achieves this by introducing an efficient mechanism for distributing objects on the web. Here is a simple example from the real world. Consider the relationship between a book publisher and a customer. Book publishers distribute thousands of copies of books to wholesale distributors and book stores. The customer purchases his book from a nearby book store instead of buying directly from a publisher. This requires spending more time and money to travel where the book is originally printed and published[4]. In a similar way, a cache server is an intermediary between the client and the web server.

Since the majority of web documents are static and cacheable, caching them can reduce the response time and the network traffic[5]. Caches generally reduce network bandwidth usage, but the specifics depend on the architecture of network and cache unit(s). A forward proxy decreases the amount of costly traffic to an external network, which is important for both Internet Service Provider on the client side and the client itself. On the other hand, a reverse proxy cache accelerates the web traffic in the origin server's network. A server-side proxy not reduces the amount of the traffic but makes the origin server considerably scalable. Different caching topologies and their properties are discussed in following sections.

Many developing countries are far behind the developed countries in terms of integrating online services. However, a great deal of work is under way to fully digitalize their systems and provide online services. Thus, we can expect Internet demands to continue to increase for the foreseeable future.

In addition to overall Internet usage, caching is also relevant for individual websites. For example, an incidence or news event might make a website or a web page suddenly very popular. This can result in the website's web server becoming a bottleneck at some critical points, with the web server being unable to respond to all of the requests. A server side cache proxy can be a the solution for such circumstances.

1.2 Motivation

When it comes to server side cache servers, there are not many choices to make. There are only a few server side reverse proxy servers available. Individuals, universities, companies, news agencies and government departments which host large or medium sized websites need reverse proxy servers. There is little independent work evaluating the performance of the available servers. In early 2000s, when the caching was a hot topic, some work was done to evaluate the performance of existing technologies. In the intervening years since then, however, despite the entrance of new products, there was almost no research comparing them. The most popular cache servers today are Apache Traffic Server, Varnish and Squid. The initial purpose of this study is to create an identical scenario for each of them and analyse their performance. Hopefully, this work make it easier for web masters to make an appropriate choice.

In reality, no web traffic traces are totally identical, and web servers have their own unique conditions. But there are common elements, such as patterns in their request streams, their content types, the mean size of their typical web page, all of which give indications of their workload. When benchmarking, all these elements will be considered so that the result will indicate which servers performs better under different circumstances.

Benchmarking cache servers is not a simple task. One reason is that the cache servers are more powerful than existing benchmarking tools. This means that

1.2. MOTIVATION

these tools are incapable of providing realistic load simulations for performance measurements. In addition, traditional web server benchmarking tools cannot measure all of the metrics relevant to cache server performance. They include end-user oriented items such as throughput and response time, but cannot measure other important items such as hit/miss rates. Finally, conventional web server benchmarking tools lack the ability to create requests according to a specified distribution function. Thus, it makes sense to try to identify the difficulties in measuring cache server performance and propose solutions.

1.3 Problem statement

This study focuses on the performance evaluation of leading open source reverse proxy servers. Varnish and Apache Traffic Server are currently the most prominent ones. The comparison will include:

- Throughput
- Response time
- Hit ratio
- Byte hit ratio
- Surrogate performance metrics

Throughput indicates how many requests a server can handle; the more requests handled, the better for the server. Response time means the mean response time as it is impossible to see the response time for each single request. Hit rate (DHR) indicates about the number of web objects or Documents that are served by the cache rather than the web server itself, while the byte hit ratio (BHR) is the number bytes served by cache.

However, measuring performance will not be limited to above mentioned metrics. In order to have a deeper understanding of cache server performance, so-called Surrogate tests will be identified to provide more comprehensive results performance evaluation. CPU, memory, disk usage and network-related metrics will be monitored through the experiments, and the data will be analysed and correlated with the benchmark results in order to identify key indicators of cache server functionality.

Finally, a stress tests will be performed on each of the servers. This sort of test is what ordinary users would most probably do in order to benchmark a cache server as the more technical benchmarking tools are too difficult for common use. Their results will also be correlated with the benchmark tool results and general performance metrics.

This research will be performed using the Polygraph [6] a benchmarking tools as well as custom scripts developed as part of this project.

1.4 Thesis Outline

In chapter 2, the background of web traffic caching is presented. Related work and methods which have been done previously will be discussed.

In the chapter 3, the methodology will presented, with discussion of different possible approaches.

In the chapter 4 , the results will be presented and analysed.

Discussion and a conclusion are the final parts of this thesis in chapter 5 and chapter 6 .

Chapter 2

Background and Related Workj

Caching has its history in the design of central processor units (CPUs) in computer architecture. CPUs are generally much faster than memory. When the CPU requests a piece of data in memory, it has to slow down to the memory's speed. Designers introduced a cache consisting of memory which is close to or within the CPU itself. When the data for a memory request is present in the cache, a cache hit, the processor does its work without slowing down its speed [8, 2]. However, when the data is not in the cache, a cache miss occurs, and the data must be fetched from the memory. In the case where there is not enough space for all the data in the cache, the new data will replace some existing data in the cache.

Web caching uses more or less the same concepts as memory caching. The major difference is that cached items from the memory system have the same type and, more importantly, the same size. On the other hand, web resources are different from object to object in terms of cacheability, type, size and retrieval cost [2].

2.1 Web servers

Web servers are entities/software on the network which provide web objects to clients. A web server might be connected to a database and make queries in order to make web objects. When a client needs a web resource, it needs to know the object's universal identifier. To do so, it needs to know which host on the internet provides that resource.

2.1.1 Static web resources

A static web resource is a set of information, e.g. web pages, which are previously generated and stored. The content is not altered due to various requests. Static web objects can be cached by a cache server since they are not dependant of time or client provided parameters. However, for security and data privacy reasons, not all static objects should be cached.

2.1.2 Dynamic web resources

Dynamic page generation technologies make it possible to generate pages at run-time based on clients' parameters [9, 10]. This gives the clients the luxury of customizing their own preferences. But the problem is that the generation of dynamic pages require processing at the server side, thus potentially creating delays in response time.

Unlike static web resources, dynamic resources are dependant on clients' requests. Even the same parameters given by the same client may result in different object generation at different times. Some solutions have been proposed to cache all dynamic pages [11, 12, 9]. However these models are not likely to function properly since there is no guarantee that future generated objects would be identical even when requested by the same client with the same parameters [9, 10].

There has been some work to find solutions so that dynamic pages can be partially cached [10, 9]. Consider a web site which requires a login. Despite the dynamic content of an authentication page, some parts of request may still be cached, such as the website's navigation bar, ads which are identical to all pages, and so on. Aninda Datta, Kaushik Dutta and coworkers investigated the idea of caching fragments or components of dynamically generated pages, and they proved that their solution, which they call DCA (Dynamic Content Accelerating), reduced the web server response time considerably [9].

Dynamic pages consist of several blocks of code which run independently. After tagging, they are stored in the buffer. Each code block belongs to a component of the page: e.g. a personalized component, a navigation component and an ad component. The HTML page will be generated by gathering all of the parts from the buffer. When parsing the request and running the dynamic script, the server is instructed to check the cache before running all the code blocks. If a code block is found in the cache, then the correspondent part in the script would be bypassed [9].

The cache management is crucial in order for caching to be effective. Datta and colleagues used a Least Likely to Use LLU algorithm in their implementation. They considered not only how recently that object has been referenced but even how likely it is to be used again by a user.

2.2 Cache servers

Cache servers are intermediary servers which store any web resource which is passed through it. It keeps a copy of object to serve any future request. Cache servers look at the HTTP headers when making the decision either to keep a local copy or not. There are generally two types of caches: browser caches and proxy caches. The browser cache is part of client's web browser; this feature is also called a client side proxy. In contrast, proxy caches are network servers (shared devices). They send requests to origin servers on behalf of the clients

[13].

2.2.1 Client side proxy

Web browsers typically store a copy of recently visited web pages. Most popular web browsers like Mozilla Firefox[14] on Linux machines, Internet Explorer[15] on the Windows Operating system, and Safari[16] on Mac OS X systems, have the caching capability. Typically, the browser stores visited pages on a part of disk which is already allocated for that purpose in the software's cache settings. The web browser verifies the objects' freshness once a session, making sure the local copy is up-to-date with the originating web server. This kind of cache is particularly useful when browsing Internet, for example, pressing the back button. However, pressing the Reload button will always fetch a new copy rather than using the old one.

2.2.2 Organization and ISP proxy caches

When using a proxy, a TCP connection is established to the proxy instead of to the content server. The proxy may be visible to the client. This is typically called a forced cache. In earlier years when dial-up connections were used by clients, establishing a connection to the external networks was both time consuming and costly. Thus, the ISP forced clients to use the ISP's cache server for external networks [13]. In such cases, the client is configured to send its requests through the proxy. The proxy in turn receives the requests, and forwards them to the server where the content actually resides. When the response has come back, the proxy saves it and forwards a copy to the client. In the future, when the same URI is requested, after checking the freshness of the cached object, it is sent to the client without bothering the origin server or any upstream caches (when multiple levels of cache servers are in use).

2.2.3 Server ISP or CDN reverse proxy caches

The ISP for a web server might want to reduce traffic bandwidth usage in its network based on the same motivations as the client ISP. While the mechanism of caching is almost the same, the name of cache is slightly different when it resides in the origin server's side. The term reverse proxy is typically used for server side caches. Reverse proxies can be implemented by the web server's ISP, the website owner or even a third party [17].

Content Distribution Networks or CDNs (e.g., Akamai [18]) play an enormous role in today's high speed internet. The competitive and popular websites want to provide their services as fast and efficiently as possible, at the lowest cost. CDNs are designed for this purpose. They are geographically spread out servers which facilitate popular web publishers' content. In addition to the typical advantages of web cache servers, which are reduced latency and network load, CDNs also increase the contents' availability [19]. If one server fails, the object

2.2. CACHE SERVERS

can still be retrieved from another server through the CDN network.

An important issue with CDNs is placement of the server so that it yields the optimal service to clients. Bo Li, Golin and co-workers designed an algorithm for this. Their goal was to place M caches among N websites. However, the results revealed that their dynamic programming algorithm didn't suit the multiple target servers case [20].

Lili, Qiu, Padmanabhan and Voelker later introduced their greedy algorithm, which achieved close to optimal results with low computational cost [21, 19]. However, the problem with this algorithm [21] and as well as older algorithms is that they may trap in local optimization. Jun Wu and colleagues introduced their generic algorithm in 2009 which alleviates this problem. The results from simulations with different rates and cost for replica placements illustrated that, except for one case for which they had the same cost, the cost for replica replacement in the generic algorithm was a half of the greedy algorithm's cost [19]. Their generic algorithm consists of a search technique for finding true or approximate solutions to optimization. It was inspired by evolutionary algorithms that are influenced by biological functions such as inheritance, mutation, selection and so on [19].

2.2.4 Server side reverse proxy cache

The main idea of having a server side cache is to make the web server scalable. A server side accelerator is not especially designed for reducing the overall traffic in the network and thus reducing bandwidth usage. Nevertheless, it can significantly improve the web servers' performance. Reverse proxy servers help to offload work from the origin server and improve the throughput [17]. This kind of cache is transparent to clients. However, the web server itself should know about the cache and be configured to listen only to the traffic coming from the front end cache server. This work has its focus on this kind of cache server. It is reasonable for medium-sized and large organizations to set up this type of proxy as they might have some peak points in their traffic patterns and the origin server might not handle all the work alone.

2.2.5 Distributed Caches and ICP

Generally when talking about a cache server, we mean one cache server. However, with today's Internet's size, a single cache may not satisfy all requests on its own. Distributed caches were designed to increase the probability of finding a hit for a document somewhere within a set of caches before going to upstream network or eventually the origin server.

Distributed caches make use of a lightweight protocol to locate objects in the cache mesh. ICP [22] is a simple protocol relying on the UDP protocol [23]. When a cache queries a neighbour cache with ICP, the response simply includes

HIT or MISS, allowing it to come back as soon as possible. A cache can not wait more than a really short time before it redirects the request to an upstream network. Thus, establishing a TCP connection for such a small query requires too much overhead. The essence of a distributed cache and caches generally is that its response should not be longer than the response without any cache.

Bruno and co-workers [17] performed a comprehensive study on the trade-offs of various distributed reverse proxy designs. In their analysis, they bounded proxy CPU and memory to origin servers, using dynamic as well as static assignment. They performed three tests with different scenarios. In the first case, where they allocated equal loads, 5 of 10 websites, to each cache node, the result showed that the CPU was the bottleneck. For the second configuration, where they assigned 9 websites to one cache node and one website to the other one, the disk became a bottleneck on the heavier loaded server. In configuration 3, all the ten websites were assigned to both cache nodes. The result revealed that the throughput at which the disk saturation point occurs in Configurations 2 and 3 increased with increasing values of α , where the α is the zipf-like distribution factor.

2.3 Cache replacement algorithms

A cache program stores its resources or objects either in a memory, on disk or both. In either case, the capacity is quickly filled. The cache software has policies related to replacing old objects with new ones. Cache replacement algorithms differ in the ways that they select an object to be deleted. The overall goal is to obtain optimized performance for cache while increasing the likelihood of cache hits [2].

2.3.1 Replacement strategies

There are different classifications of replacement policies. Aggarwal [24] proposed a three type classification: direct extensions of traditional strategies, key-based replacement strategies and function-based replacement strategies. In 2001, Krishnamurthy and Rexford [25] proposed another scheme which classified algorithms according to their complexity: one-level strategies that used one factor, two level strategies that uses a primary factor and secondary factors and weighted strategies that combines multiple factors. Factors which are important for cache algorithms are recency, frequency, size, cost of fetching, modification time and expiration time [26].

There are however two problems with the algorithms cited above. First, since traditional algorithms also use a key (factor), the first two classes can be combined in one. Second, randomized policies can not be included in any of the classes above described.

Jin and Bestavros [27] created a strategy classification based on recency, frequency and recency/frequency. This has the benefit that it distinguishes between recency and frequency, which are the most considered factors, but it still

2.3. CACHE REPLACEMENT ALGORITHMS

lacks the classification of randomized strategies. It in addition, has problems with algorithms which do not use frequency and recency.

Considering previous works, [24, 25, 27] Podling and Laszlu [26] proposed their classification which comprehensively covers most existing policies.

Recency based policies

These strategies consider recency as the main factor. LRU (least recently used) is the most popular implementation of this strategies. All other variations are more or less extensions to LRU. LRU looks at the locality of objects. There are two kinds of locality, temporal and spatial. Temporal locality looks at the time the object was last referenced. Spatial locality, on the other hand, considers known object access patterns and extrapolates them to other objects. Locality of object characterizes the ability to anticipate future calls for objects from the past calls.

Some of the more important implementations of recency-based strategies are:

- LRU: this strategy removes the least recently used object.
- LRU-Threshold: in this strategy, an object is not cached if S_i (the size of object i) is larger than a given threshold[28].
- Pitkow/Recker's strategy uses LRU, but differentiates between objects by their size. For example, for objects that are requested on the same day, the object with largest size will be removed first[29].
- EXP1 uses the current time and last time the object was accessed to weight the importance of the object[30].
- Value-Aging uses an 2.1 formula. $V_{new}(i)$ is updated each time the object is requested, according to the formula below (where C_t is the current time)[31]:

$$V_{new}(i) = V_{old}(i) + C_t * \sqrt{\frac{C_t - t_i}{2}} \quad (2.1)$$

- HLRU records the number of times a web page has been referenced (its access history). It defines a hist function where $hist(x,h)$ is the time of the past h th reference to a specific cached object x . HLRU evicts the object with the maximum hist value. Value $hist=0$ uses the original LRU strategy[32].
- PSS(Pyramidal selection scheme): this strategy builds a pyramidal classification of objects depending on their size. All objects of class i have the size between 2^{i-1} and $2^i - 1$. Therefore there are N different classes where $N = \log_2(M + 1)$ where M is the cache size. Each class has an LRU list. When choosing an item for replacement, PSS compares the least recently used object of all classes. The object i with largest value of $s_i \Delta T_i$ of all lists will be evicted[24].

2.3. CACHE REPLACEMENT ALGORITHMS

- LRU-LSC: this strategy takes advantage of LRU and finds out the activity of objects. When replacing, objects with less activity will be moved to a second list as long as the size of the new list is less than a given threshold θB where the θ is the threshold parameter and B is total cache size. Objects are removed from the new list until the accumulated size of them subtracted from a total cache size is less than a specific value[33].
- Partitioned caching: according to this strategy, the cache is divided into the parts: small, medium and large. Each part keeps its own lists and implements LRU. The thresholds for this classifications are derived from previous web traces. Assume S_c is total size of cache and S_{c1}, S_{c2} and S_{c3} are cache sizes for classifications 1, 2 and 3, with $S_c = S_{c1} + S_{c2} + S_{c3}$. Mutra et. al. [34] experimentally showed that $S_{c1} < S_{c2} < S_{c3}$ should hold[34].

Frequency-based policies

These algorithms focus on the number of time an object has been referenced. The more frequently an object has been referenced, the more probable it is to be requested in the future. The best implementations of this policy is LFU (least frequently used). Many other implementations are extensions of LFU. There are two main design considerations for LFU perfect LFU and in-cache LFU.

Perfect-LFU takes in account all times an object in referenced, even if it has been evicted from the cache. The next time when it comes in to the cache, there are already counters exist for that. This method gives a better overview and understanding of the traffic. Its drawback however is greater CPU and disk usage.

In-cache LFU performs the same as perfect-LFU, but it resets the counter each time an object is evicted from the cache.

In the following strategies LFU means In-cache LFU.

- LFU removes the least frequently used object.
- LFU-Aging: Objects which were popular in a previous period will be kept in cache even though they have not been not used for a long time time. This is don because of their high frequency rank. In order to reduce this cache population, an effective age threshold is introduced. When the average value of all frequency counters breaches the the threshold, LFU-Aging divides the counters by 2.[35].
- LFU-DA: The problem with the previous strategy (LFU-Aging) is that it is highly dependant on the threshold value and other parameters like maximal frequency value. LFU-DA solves this problem with calculating a new value, K_i for object i . $K_i = f_i + L$ where L is an aging factor that starts at zero. LFU-AD chooses the objects with smallest value of K_i and then this value is assigned to L [35].

2.3. CACHE REPLACEMENT ALGORITHMS

- α - Aging: This is an explicit aging method with a periodic aging function[31].

$$f(v) = \alpha * f_i \quad 0 \leq \alpha \leq 1 \quad (2.2)$$

- sw-LFU (server weighted LFU): this policy makes use of a weighted frequency counter. The w_i weight of object i shows how much the server of i appreciates caching of that object i . Thus, the server can affect the caching of the object[36].

Recency/Frequency based strategies

These strategies use both recency and frequency as the decisive criteria for selecting objects for replacement. Depending on the design there might be other factors involved to make the best decision.

- SLRU [35, 26]: This strategy divides the cache space in two parts: one protected, which is specified for popular objects, and another unprotected part. Both parts implement LRU. When an object is fetched to the cache, it will first be placed into the unprotected segment. After the first hit, the object will be transferred to the protected part. Evicted objects are chosen only from the unprotected segment. Objects are moved back to the unprotected part, as the most recently used item, when space is needed in the protected part.
- Generational replacement: In this method, all objects are listed in n lists ($n \geq 2$). Each list $i < n$ contains objects which have been called i times. List n contains all objects with n or more calls. A request for an object deletes it from its current list and inserts it into the next list (at the beginning). In this method, objects are inserted at the beginning of the list and are deleted from the end of the list[37].
- LRU*: All the objects are stored in a single LRU list, and each has a counter. When there is hit for an object, it will be moved to the top of the list, and its counter increments by one. At each replacement, the counter of least recently used object is checked. If it is zero, the object will be removed; if not, the counter will be decreased by one and the item will be moved to the beginning of the list[38].
- LRU-HOT maintains two lists: one hot list for popular objects and one cold list for less popular objects. An object is considered hot if its frequency counter is larger than a threshold set on the server. This information is sent along with the object to the client/cache server, and the object will be inserted to the corresponding list. This technique maintains two counters: a base counter which increments after each request, and a hot object counter which increases after each α . When an object is requested, it is stored at the beginning of the corresponding list and assigned the actual value of base counter. When a replacement is needed, the values of the last two objects in both lists are recalculated and the object with the smallest value is discarded[39].

2.3. CACHE REPLACEMENT ALGORITHMS

- HYPER-G [40], CSS(Cubic Selection Scheme)[41] and LRU-SP [42] are other strategies which are more or less similar to above methods.

Function based strategies

These strategies use a general function to compute the value of an object. They generally choose the object with smallest value. The following are most popular implementations of function based strategies.

- GD(Greedy Dual)-Size algorithm[43]: It maintains an identifying value H_i . After each request or hit, the H_i is recalculated by

$$H_i = \frac{c_i}{s_i} + L \quad (2.3)$$

where s_i and c_i are size and cost of retrieval for object i and L is a run aging factor which like LFU-DA starts from zero. The GD-size algorithm chooses the object with the smallest value and assigns this value to L .

- GDSF Greedy Dual Size Frequency[35] is similar to GD-size but calculates the H_i by

$$H_i = f_i * \frac{c_i}{s_i} + L \quad (2.4)$$

where f_i is the frequency of the object. This was more generally proposed by Cherkasova and Ciardo[5] as:

$$H_i = \frac{f_i^\alpha}{s_i^\beta} + L \quad (2.5)$$

where α and β are weighting factors and C_i is set to 1 .

- GD* The function for calculating H_i is:

$$H_i = \left(\frac{f_i * c_i}{s_i} \right)^{\frac{1}{\beta}} + L \quad (2.6)$$

Here, β is the weighting factor which characterizes reference correlation via the distribution of reference intervals for objects with the same popularity[27].

- Other strategies Server assisted cache replacement[44], TSP(Tailor Series Prediction)[45], MIX[46], HYBRID[47] , LRV[48] and LUV[49] are other implementations of function based strategies.

Randomized strategies

These strategies use randomized decision making to find an object to remove. The following are the major randomized algorithms.

- RAND: This strategy simply chooses a random object to remove.
- HARMONIC [33]: Unlike RAND, which uses an equal probability for all objects, HARMONIC uses a probability inversely proportional to its cost, where $\text{cost} = \frac{c_i}{s_i}$
- LRU-C and LRU-S[50]: LRU-C is a randomized version of LRU. It defines $c_{max} = \max\{c_1, c_2, \dots, c_N\}$ as the maximum access cost of all N objects, and $\bar{c} = \frac{c_i}{c_{max}}$ as the normalized cost of the object i. When the object i is requested, it will be moved to the top of the cache with the probability of \bar{c} ; otherwise, nothing happens.
LRU-S is similar but it uses size instead of access cost. $s_{min} = \min\{s_1, s_2, \dots, s_N\}$ is the smallest size among all N objects, and $d_i = s_{min}/s_i$ is the normalized density of the object i. LRU-S acts as LRU with d_i as probability; otherwise, nothing happens.
- Randomized replacement with general value functions[51]: In this method, N objects are drawn from the cache and the least used one is discarded. Any utility function can identify usefulness of the objects. After removal of the chosen object, the rest of the objects M(M < N) are retained in the memory. In the next replacement, however, M-N objects are drawn from the cache and the least useful of all collection (including M from before and N-M new objects) is removed.

2.4 HTTP

Caching of individual pages is controlled to some extent by the information contained in the HTTP headers of the web documents.

The Hyper Text Transfer Protocol is the most used web transport protocol. The intention of Tim Berners-Lee and his co-workers in the HTTP group was to design a simple lightweight protocol [4] specifically for the web. HTTP is an application-level protocol which is used between distributed, collaborative, network-based and hypermedia information systems RFC2068 [52]. It relies on the Uniform Resource Identifier, via the URI Standard RFC3986 [53].

Since the early 1990s when the first version of HTTP was designed, there have been three revisions of the protocol. The first one, retroactively called HTTP/0.9, was a very simple protocol that lacked many basic features which are relied on today. In 1996, HTTP/1.0 was designed. It had a small set of features but still kept the simplicity of the design. However, with the large growth of web, web developers quickly discovered that HTTP/1.0 could not provide all the functionality needed for the new web services [4].

The HTTP working group of Internet Engineering Task Force worked long and hard to evolve the initial simple protocol into a complicated protocol which

2.4. HTTP

features that could handle the most of the requirements. Persistent connections, better cache control, content negotiations and range requests are important features which were added to HTTP/1.1 [4].

2.4.1 HTTP Message Structure

HTTP uses a well-defined message structure. An HTTP message may be either a request or a response. Either way, it consists of two parts: the header and the body. Every HTTP message must have a header, but having a body is optional.

Example HTTP Request Header

HTTP header

```
GET /http.html Http1.1
Host: www.example.com
Accept: image/gif,image/x-xbitmap,image/jpeg,image/pjpeg,
Accept-Language: en
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Connection: Keep-Alive
```

Example http Response Header

```
HTTP/1.1 200 OK
Date: Mon, 12 Mar 2011 19:12:16 GMT
Server: Apache/1.3.12 (Unix) Debian/GNU mod_perl/1.24
Cache-Control: max-age=3600, must-revalidate
Last-Modified: Fri, 22 Sep 2010 14:16:18
Accept-Ranges: bytes
Content-Length: 3369
Content-Type: text/html
```

Message Headers

The syntax of a header is a series of attributes in the form of Name:Value pairs. Multiple values are separated by commas [4, 52]. HTTP defines four categories for headers: entity, request, response and general. Some headers are only specified for request messages while others are defined only for responses. For example, Host and If-Modified-Since are only request headers. Using such headers in responses would be meaningless. On the other hand, Date, Last-Modified and Expires are response only headers.

Cache-Control Headers The Cache-Control general header field specifies directives that MUST be complied by caching mechanisms for request/response chain. There are two subtypes of Cache-Control directives: cache-request-directive and cache-response-directive [52]. Each of them supports several attributes which are listed in table 2.1.

Header : Cache-Control	
cache-request-directive	cache-response-directive
no-cache	public
no-store	private
max-age	no-cache
max-stale	no-store
min-fresh	no-transform
no-transform	must-revalidate
only-if-cached	proxy-revalidate
cache-extension	max-age, s-maxage cache-extension

Table 2.1: The Cache Control directives

Here is a short explanation of each directive taken directly from the RFC. A full description of all of the headers is found in RFC2616 [54].

- no-cache: If the no-cache directive does not specify a field-name, then a cache MUST NOT use the response to satisfy a subsequent request without successful revalidation with the origin server
- no-store: The purpose of the no-store directive is to prevent the inadvertent release or retention of sensitive information
- max-age: Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds
- max-stale: Indicates that the client is willing to accept a response that has exceeded its expiration time, but not the max-stale specified time
- min-fresh: Indicates that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the specified time in seconds
- no-transform: the cache or proxy MUST NOT change any aspect of the entity-body that is specified by these headers, including the value of the entity-body itself
- only-if-cached: In some cases, such as times of extremely poor network connectivity, a client may want a cache to return only those responses that it currently has stored, and not to reload or revalidate with the origin server. To do this, the client may include the only-if-cached directive in a request
- cache-extension: The Cache-Control header field can be extended through the use of one or more cache-extension tokens, each with an optional assigned value. Informational extensions (those which do not require a

2.4. HTTP

change in cache behaviour) MAY be added without changing the semantics of other directives

- **public**: Indicates that the response MAY be cached by any cache, even if it would normally be non-cachable or cachable only within a non- shared cache
- **private**: Indicates that all or part of the response message is intended for a single user and MUST NOT be cached by a shared cache
- **must-revalidate**: Because a cache MAY be configured to ignore a server's specified expiration time, and because a client request MAY include a max- stale directive (which has a similar effect), the protocol also includes a mechanism for the origin server to require revalidation of a cache entry on any subsequent use
- **proxy-revalidate**: The proxy-revalidate directive has the same meaning as the must- revalidate directive, except that it does not apply to non-shared user agent caches
- **s-maxage**: If a response includes an s-maxage directive, then for a shared cache (but not for a private cache), the maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header

The headers Cache-Control, Expires, If-Modified-Since and Last-Modified are most important for caching. The Cache-Control header specifies whether the object is cacheable or not, with the diverse values they can get.

The first line in a message is special. In a request, the line called request line and includes a method, a uri and a HTTP version.

$$\underbrace{GET}_{\text{method}} \ / \ \underbrace{\quad}_{\text{uri}} \ \underbrace{HTTP/1.1}_{\text{version}} \quad (2.7)$$

Table 2.2 lists the different HTTP methods [52, 4].

2.5. CACHING SOFTWARE

Method	Description
GET	A request for information identified by the request URI
HEAD	HEAD is identical to GET but without a body
POST	A request to server which requires it to process information in the message
PUT	A request to save the attached body to the URI
TRACE	A loopback method which is useful for testing proxies between client and server
DELETE	A request which requires server to eliminate a named URI
OPTIONS	A request for information about the server's support for optional features
CONNECT	Used to tunnel certain protocols via a proxy

Table 2.2: HTTP protocol methods

Message Body

The body of the message is optional, and the content which is going to be transferred appears in this part.

2.5 Caching software

2.5.1 Apache Traffic Server

Apache Traffic Server(ATS) [55] is a fast, scalable and extensible HTTP/1.1 compliant caching proxy server. It was originally a Yahoo product which was donated to the Apache software foundation[56]. Traffic Server is a high performance web proxy-caching server, and it has a robust plug-in API that allows users to modify and develop its behaviour and abilities. From the beginning it was designed as a multi-threaded event driven server, and therefore scales very well on modern multi-core servers. Its native support for dynamically loading shared objects makes it to interact with the core engine. Apache proxy server is a generic implementation that can be utilized to proxy and cache a variety of workloads, from single site acceleration to CDN deployment and very large ISP proxy caching, and it includes features like partitioning of the cache.

Structure and components

Traffic Server consists of five components. They are:

- The cache(object store database), the object store data base indexes the objects using their URLs and the headers.Then using the policies, store both small and large objects efficiently.
- The Ram cache : maintains extremely high ranked objects to off-load the disks under heavy peaks.
- The host database: maintains the DNS entries of origin servers along with HTTP version of each host and health of backends.

2.5. CACHING SOFTWARE

- DNS resolver: Traffic Server includes a fast DNS resolver that issues directly DNS commands rather than using traditional resolver libraries, which leads to parallel and faster commands.
- Traffic Server Processes: TS includes three processes :
 - traffic_server is the main engine which receives, indexes, caches and serves the requests.
 - traffic_manager which takes care of traffic-server. It can monitor, reconfigure and launch the traffic_server process. In case the traffic-server failes, the traffic-manager makes a FIFO queue from the incoming requests while restarting the traffic-server process.
 - traffic_cop is responsible for health of both traffic_server and traffic_manager. It sends heartbeat requests in small intervals to both processes. If there are no responses, that restarts the traffic_manager and traffic_server processes.

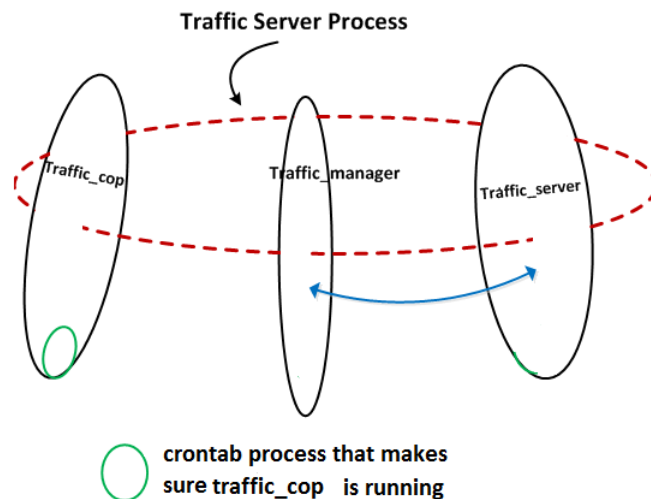


Figure 2.1: Apache TrafficServer processes

Replacement policy

Apache Traffic Server uses a combination of several policies to evict the objects from the Ram. It uses all the LFU, LRU, CLOCK, GDFS and 2Q together. It is called CLFUS (Clocked Least Frequently Used by Size).

It maintains two lists of objects. The Cached List includes the actual pages in the memory. The new objects are inserted in a FIFO queue

with a LRU policy. When there is a hit from the list, the hit object is reinserted to the top of the list. The History List keeps a list of objects that at least once have been requested. Each CLOCK the list is dequeued and the least recently used object is deleted if the hit field of this list is not greater than 1[57].

The policy implemented for disk object eviction is a simple FIFO, but some minor policies are also implemented to not blindly throw the objects out of the cache, like

2.5.2 Varnish

Varnish [58] is free software licensed under a two-clause BSD licence, also known as the FreeBSD licence. The project was initiated in 2005. Its first version was released in September 2006 (Varnish cache 1.0), and the latest version Varnish cache 3.0.2 was released in October 2011. The key features of Varnish are its performance and flexibility. Via its own configuration language vcl (Varnish configuration language), it is highly configurable, and users can make policies how to handle various traffic scenarios[58].

Structure and policy

Varnish runs a parent and a child process. When starting Varnishadm daemon, The parent will start the child process and when the child process dies for any reason the parent will recover the process again.

In the Varnish's code there are subroutines which running the process. The subroutines vcl_recv, vcl_fetch, vcl_pipe, vcl_pass, vcl_hit, vcl_miss, and vcl_error are the most used ones. The vcl_recv and vcl_fetch alone can handle the most part of customized VCL files.

- vcl_recv: receives the requests, parse them, makes decision of serving from the cache or a backend etc. It is able to alter the headers as well.
- vcl_fetch: This method is called when an object is retrieved from a backend. The basic operations here are to change the header, change the backend if previous one was unhealthy etc.

When it comes to replacement strategies, Varnish does not know objects are on the disk and what objects are in the RAM. It implements a single global LRU covering both of them.

To put it all together Table 2.3 shows the different policies implemented by Apache Traffic Server and Varnish server for replacement of pages.

2.6. CHALLENGES

	RAM Policy	Disk policy
Apache Traffic Server	CLOCK, LRU,LFU, GDFS,	FIFO
Varnish	LRU	LRU

Table 2.3: Replacement policies for RAM and Disk, Varnish vs ATS

2.5.3 Others

Other known reverse proxies are Nginx, Perlbal, pound, lighttpd, HAProxy, MacAfee web gateway, Citrix Systems Netscaler, aiCache. Most of these products are free and have the capability of load balancing as well.

2.6 Challenges

The vastness of Internet and the large number of elements involved in the http traffic, makes it quite challenging to deal with the http traffic, especially in a cached environment which introduces more complexity. There are some main challenges which are discussed in the following sections.

2.6.1 Realistic Workloads

In order to measure the performance of a Traffic Server, it should be exposed to a stream of requests and the real web servers behind that. The workload however should be reproduceable in order to run several experiments and verify the correctness of the results. A traffic pattern seen from a reverse proxy's point of view is quite different from a forward proxy's view. The reverse proxies cache traffic from specific and possibly a few number of websites while the forward proxies generally cache the traffic from any number of websites.

When benchmarking an origin server mostly throughput and bandwidth are considered as the main factors. The hit ratio and the object freshness are not taken care of, while those two are important factors in web cache environment. The challenge is produce and measure the traffic to address these factors.

Zipf's Law distribution model

Zipf's Law states that the relative probability of a request for the i 'th most popular page is proportional to $1/i$ [59]. In other words Zipf's law, is the observation that frequency of occurrence of some event (P), as a function of the rank (i) when the rank is determined by the above frequency of occurrence, is a power-law function with the exponent α close to unity.

$$P_i \sim 1/i^\alpha$$

An important characteristic of a request trace to a cache follows a Zipf's Law[59, 60, 61] or Zipf-like Law [62, 59]distribution. Zips's Law is a mathematical model

2.6. CHALLENGES

which states that a specified number of elements have a high probability score and an average number of elements have a middle probability and a large number of elements have a small probability of occurring.

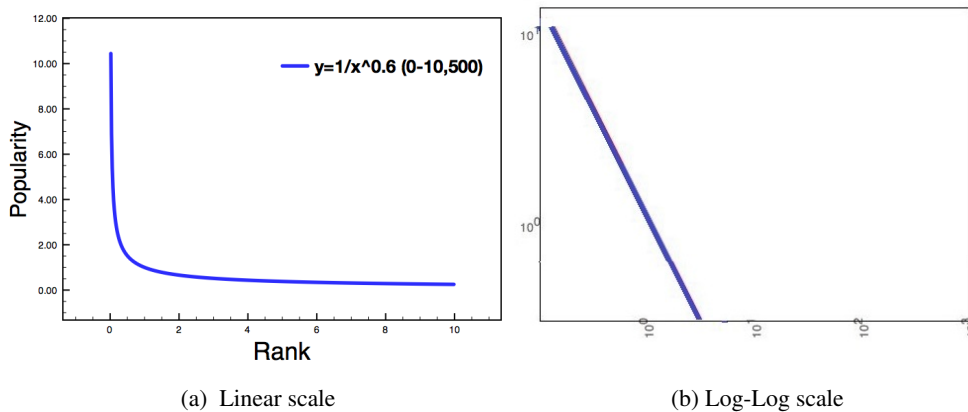


Figure 2.2: Popularity of web objects follows the Zipf-like Law with α close to 1. Figures illustrate a Zips-Like law distribution with $\alpha=0.90$

The probability of occurring one event is proportional to its ranking. As the rank grows the occurrence of event falls dramatically in the top part of ranking table.

The generated workload and the popularity distribution of objects should satisfy the characteristics of a real trace which follows the Zipf-like law. That needs lots of accurate calculations. In the generated workload the unique objects and their ranks should be kept track.

The problem in the simulated environment is the time. A test can not last as long as a real scenario, therefore the time should be "compressed". This has side effects on the other characteristics of the workload. One the side effects is on the Zipf-like popularity model. As in the Zipf law the "hot data set" is small and easily can reside in the memory. That causes a big percentage of "memory hits". The caches mostly depend up on their disk systems, and in this case due to high memory hits the cache's disk hits are not pushed enough [6, 1].

Figure 2.3 which is taken from cache-off's web site shows how small a "hot data set" can be in zipf like trace compared with a uniform model.

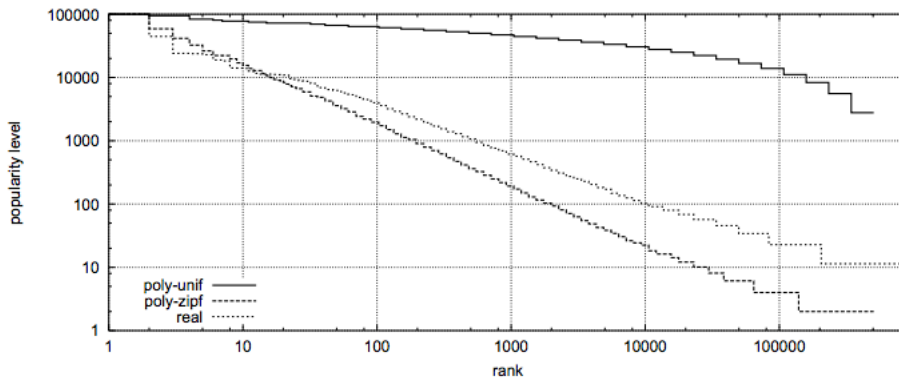
Figure 2.3: Zipf versus uniform popularity models, taken from *Cacheoff* [1]

Figure illustrates that there are only 10 or more URLs which have been requested more than 10000 times in the zipf model while 100000 or more URLs have been requested 10000 times or more in uniform model. Each trace contains 1.5 million URLs. The real trace is taken by [1] from `sv.cache.nlanr.net` and the poly-unif and poly-zipf are generated by polygraph tool which comes in the following section.

2.6.2 Lack of recent works

In late 90s and early 2000s, the web traffic caching was a hot topic. There exist quite many works in this area from those times. However after introducing the CDNs or Content Delivery Networks the main focus of work turned to CDN caches rather than proxy caches. However during this period Internet has grown rapidly and the tendency shows the growth is still continuing in high tempo. That means the characteristics of web traffic has changed accordingly and understanding the new traffic and its characteristics is important.

A number of Cache-Off benchmarks was performed in early 2001. Unfortunately there are no recent works and enough documentations to identify important elements and their significance according to today's web traffic when it comes to reverse proxy caching. Besides, Benchmarking forward proxies is different from benchmarking reverse proxies, and nearly all of work done in Cache-Offs are about forward proxies.

2.6.3 Tools specifically designed for cache benchmarking

Benchmarking a cache is not something which is done very often and by individuals. There are quite a good number of web server benchmarking tools. Unlike web servers, for cache servers the choice is very limited. There are only a few number of open source tools exist. In the next section the most popular ones and their capabilities are shortly discussed. As mentioned previously the proxy caching topic was popular in early 2000s and most of today's existing tools were developed then. Web polygraph is the only one which came with

2.6. CHALLENGES

updates afterwards.

WebJamma and proxysizer There are however a small number of tools which can read from a real trace and play back http access log files. WebJamma and proxysizer can generate workloads by reading existing traces and log files.

Winconsin is a tool that initially was made to benchmark the proxy servers. That is able to generate a very simple workload and supports the concept of server and client processes. However this tool was not updated afterwards and does not support many of new features in a web traffic trace. Among others HTTP/1.1 which is most commonly used protocol on the web traffic is not supported by the Winconsin.

Web polygraph is a powerful tool which claims to be able to generate a variety of workloads. It supports HTTP/1.1 and have got many features which enriches the load generated with almost all the needed characteristics. It simulates both clients and servers as well as generating the requests.

Chapter 3

Model and Methodology

In this chapter, the approach, test environment, benchmarking tools and basic cache server configuration parts will be discussed:

- Section 3.1 discusses the possible and proposed approaches to the problem.
- Section 3.2 describes the test environment including hardware specifications and network configuration.
- Sections 3.3 and 3.4 describe the tools which were used for this work:
 - Web Polygraph
 - blktrace
 - Seekwatcher
 - Custom scripts
- Sections 3.5 through 3.8 describe the configuration specifications for Varnish, Apache Traffic Server and Polygraph.
- Finally, Section 3.9 describes the various workloads designed for this experiment.

3.1 Approach

There are two dimensions that are considered in the proxy evaluation architecture: the implementation environment and the source of workload. The possible workload sources divide into three categories: artificial, captured logs and current requests (the workload source space). Similarly, three main algorithms are available for the evaluations; simulated system and networks, real system/isolated networks and real system/real networks comprise the implementation space. This is illustrated in Figure 3.1

In an ideal test model, the following characteristics are desirable in order to get the best possible results:

- Reproducibility

3.1. APPROACH

- Flexibility of testing
- Observability of direct results
- Testing based on real traffic
- Testing performed on real systems

The first priority for this work is reproducibility. It is important to be able to reproduce tests as needed for both the Varnish server and Apache Traffic Server. In addition, the tests should be reproducible across runs in order to confirm the results. Furthermore, other people who might want to repeat the experiment should be able to do so.

Flexible testing is important because it provides the opportunity to produce scenarios that are interesting. Thus, by changing some characteristics of the configurations it is possible to create a variety of tests which can figure out how different variables affect the results.

Tests should be designed so that the results produced are direct measures of the behaviour and performance of the servers being tested.

Testing on real systems and real networks is also desirable because these environments give the most realistic results. Therefore, a scenario with real traffic on real systems would seem to be ideal. However, such a situation is not reproducible both because the state of a real scenario changes over time, and because real networks are inherently chaotic. A real system on an isolated network is a very good alternative. Since the main purpose of this work is to focus on performance of two products, an isolated network eliminates all the variations and unpredictability of a real network.

Doing the experiment with the real traffic again gives the most realistic results as it comprises real traffic patterns and the exact sizes of documents and their associated cost of retrieval. However, the main constraint in this case is again the reproducibility of the test. A good alternative which provides reproducibility is using traffic generated from the captured logs of real servers running in production. Captured logs maintain all the characteristics of live traffic. Furthermore, such replayed traffic is reproducible. The problem, however, is the flexibility of the trace; captured logs cannot be tuned to assume whatever traffic characteristics become desirable as the experiment proceeds. The other issue with captured logs is the validity of objects, due to the fact that not all the objects in the captured log trace are still valid or in existence at the time the trace is used for testing. An artificial workload does not have any of these problems.

3.2. TEST ENVIRONMENT

		WorkloadSource		
		Artificial	Captured Log	Current Req.
simulated systems/ Networks	Reproducible	Y	Y	Y
	Real traffic		Y	Y
	Observable	Y	Y	Y
	Flexible testing	Y		
Real systems /isolated network	Reproducible	Y	Y	
	Real traffic		Y	Y
	Observable	Y	Y	Y
	Flexible testing	Y		
Real systems/ Real network	Reproducible			
	Real traffic		Y	Y
	observable	Y	Y	Y
	Flexible testing	Y		

Figure 3.1: Different possible experimental approaches

Figure 3.1 summarizes the different possibilities for the generation of workload and implementing the test systems. It indicates which scenarios can satisfy the various requirements. The green highlighted area is the approach selected for this work

The artificial workload is reproducible, flexible and generates valid objects. The disadvantage of using an artificial workload is its not being real. However, there is advanced and complicated softwares which can produce close to real traffic to alleviate this problem. It can produce the long-tailed distributions of object size and object popularity which are characteristic of real traffic.

Thus, the selected method for this work is the scenario with real systems (as they are the most realistic), but with an isolated network to eliminate the variations from the Internet. The focus is to compare the performance of the proxies in the most realistic but identical scenario that is possible. The testing workload is close to realistic despite the fact that it is generated artificially. The workload also satisfies the reproducibility, flexibility and other requirements.

3.2 Test environment

Three computers were used for this work, connected to each other by a Gigabyte L2 switch. The specifications for machines summarized in Table 3.1.

3.2. TEST ENVIRONMENT

Manufacturer	DELL Optiplex 745
OS	Debian: kernel 2.6.32
RAM	4GB DDR2
Disk	2 x 80 GB 7200 RPM SATA
CPU	Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz
Network Card	PCI Gigabit NIC and 5754 Gigabit Ethernet

Table 3.1: Test hardware specifications

Figure 3.2 shows the network setup. All three machines are directly connected to the Gigabyte switch. Two of machines have 2 network cards with two IP addresses. The private network 192.168.0.0 addresses with netmask 255.255.255.0 were used.

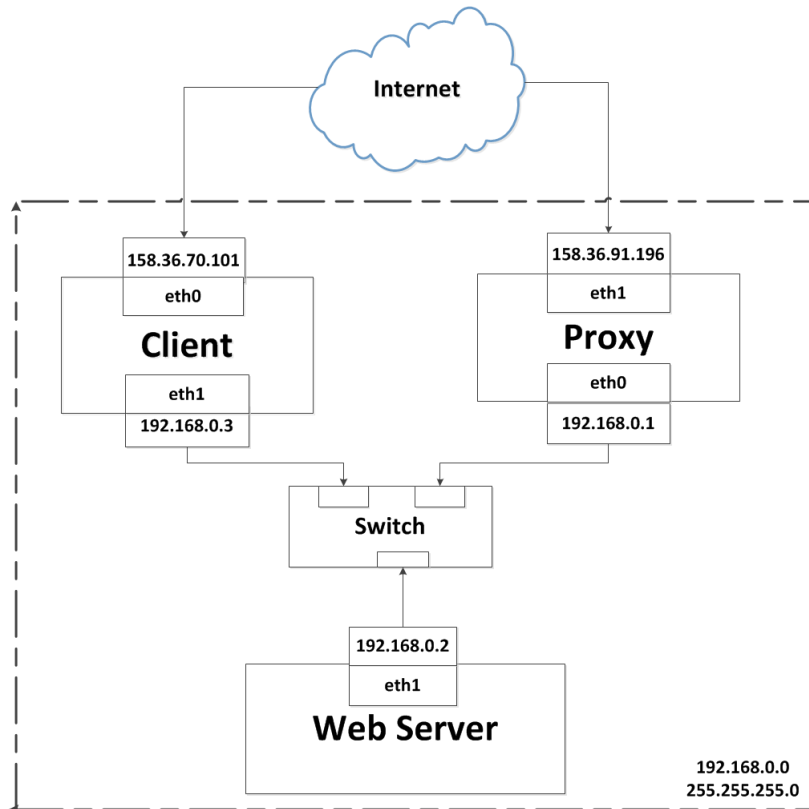


Figure 3.2: Experimental private network setup

The figure shows how the cache server, the web server and the client machine are connected together and to the Internet, as well as the IP addresses assigned to each machine. The cache server is where the Varnish and Apache Traffic Server caching servers will run, the web server is the back end machine which the caching proxy serves, and the client machine is the source of the network traffic and where the experimental results will be recorded.

3.3. WEB POLYGRAPH

Machine/Interface	eth0	eth1
Cache server	192.1680.1	158.36.91.196
Web server	n/a	192.168.0.2
Client	128.39.75.101	192.168.0.3

Table 3.2: IP addresses

The web server machine which does not have a public IP address; it gets the Internet from via the cache server machine. The ssh traffic destined for the cache server machine's port number 222 is forwarded to the web server. Here are the iptables rules which forward the traffic to and masquerades traffic from the web server machine.

```
iptables

iptables -t nat -A PREROUTING -i eth1 -p tcp --dport 222 -j DNAT --to-destination 192.168.0.2:22

iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

These iptables rules do not persist across system reboots. The following commands rewrite the iptables rules after each restart. The rules are stored in the file `/etc/firewall.conf`, and a shell script is run by the `ifup` command when the interface restarts, so that the rules are installed again. For this to work, port forwarding for the system must be enabled (first command below), and then the firewall configuration can be saved to a text file:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables-save > /etc/firewall.conf
```

The simple shell script `iptables`, located in `/etc/network/if-up`, is used to restore the rules from `firewall.conf` on reboot:

```
/etc/network/if-up/iptables

#!/bin/sh

iptables-restore </etc/firewall.conf
```

3.3 Web Polygraph

Web Polygraph [6] is the main tool will be used for benchmarking in this work (referred to hereafter as simply Polygraph). Polygraph is a freely available tool which is used for performance testing. Polygraph is de-facto industry standard

3.3. WEB POLYGRAPH

for testing HTTP intermediaries. Testing of proxy caches, origin server accelerator and L4/7 switches, content filters and other web intermediaries can be performed using this tool. Its most important features are

- High-performance HTTP clients and servers
- Realistic HTTP traffic generation
- Flexible content simulation
- A powerful configuration language

The plan for this research was to use an existing tool rather than writing such a tool from scratch. The main focus was to be on observing cache server behaviour rather than developing a sophisticated tool. As discussed in Chapter 2, Polygraph is best and only choice for this purpose. It can generate the desired load patterns needed for this research. It is also generally up-to-date with current caching software, and it also supports more features than other software. This tool was chosen despite the fact that the significant start up overhead would be significant, which turned out to be even greater than initially anticipated. The difficulties and bugs with this tool are discussed in Chapter 5.

Polygraph can simulate a large number of clients and server processes provided that system resources support them. The main bottleneck is limited memory, as the process will die if it runs out of the memory. The other limited resources are CPU capacity, the number of TCP ports and network bandwidth. Polygraph use a custom configuration language called PGL which must be used to define workload parameters and characteristics.

Polygraph works by creating a set of client processes that it calls robots. Each robot (client) is able to behave like a real web user and can request a variety of web object content types with predefined distribution probabilities.

Polygraph runs two type of processes: `polygraph-server` and `polygraph-client` which simulate clients and servers on the network (respectively). The generated traffic from the `polygraph-client` is sent to the `polygraph-server` through the proxy.

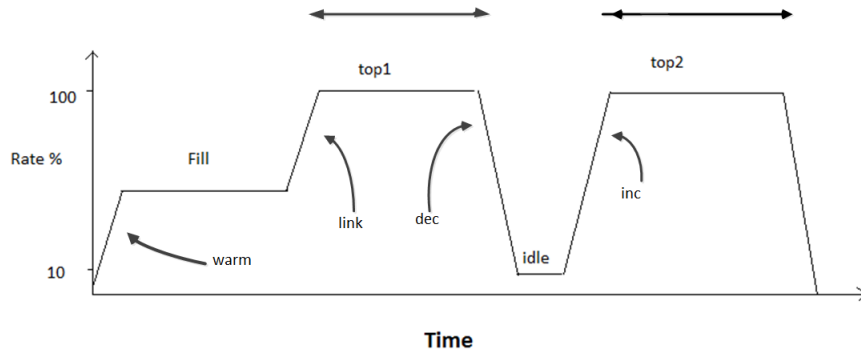
3.3.1 Polygraph testing phases

Polygraph tests are accomplished via a series of phases. These phases are configured using PGL, and there are a variety of parameters available for doing so. For example, each phase has its own begin and end request rate factors. After reaching to each phase's goal Polygraph automatically moves to next phase.

The Polygraph phases `warm`, `fill`, `inc`, `top1`, `dec`, `idle`, `top2` and `cool` are the phases used for reverse proxy benchmarking, as illustrated in Figure 3.3.

3.3. WEB POLYGRAPH

Figure 3.3: Phases



The different phases are defined as follows:

- warm: In this warm-up phase, the client and server robots try to find each other and negotiate their configurations in order to detect any problems.
- fill In this the phase, the initially empty cache is filled by the client robots. The time of filling varies depending on the size of the cache, the request rate, and the recurrence rate of requests. To obtain the best results, the goal of this phase is to fill twice the size of the cache in order to reach a steady state. In this experiment, the recurrence rate is set to 5% to speed up the filling.
- link: The purpose of this phase is to prevent a sudden increase in the request rate. In this phase, which links the fill phase and the first main testing period, the request rate is gradually increased to the desired rate for the following top1 phase (the peak rate for the test).
- top1: This key testing phase runs at the maximum specified request rate. The rate is kept at the constant peak level during this phase. In order to focus on caching behavior, the recurrence of objects is set to a high value (95%) to prevent clients from generating new objects.
- dec: This phase decreases the request rate gradually down to the level selected for the following idle phase, which is typically 10% of the top1 peak rate.
- idle: During this phase, the request rate is set to a very low level in order to compare the performance under low and heavy traffic conditions. In this phase, the proxy server should have the time to do its regular off-line activities such as lazy garbage collection.

3.3. WEB POLYGRAPH

- inc: This phase is similar to the link phase. During the inc phase, the request rate is gradually increased from 10% to 100% of peak rate, in preparation for the following top2 phase.
- top2 This phase is the most important phase. Comparing the results from top1 and top2 shows if the proxy is truly in a steady state. When the test reaches this phase and the proxy server has reached a steady state, it has experienced both high and low request rates requests. The results from this phase is are used for comparison with the previous high request rate phaser, top1.
- cool Once the testing is finished, the robots and servers will cool down from the peak request rate to zero during this phase.

3.3.2 Command line options

Polygraph has a large number of command line options. Some of them can also be specified in the configuration files. When running Polygraph for this test, the following options are used:

- `-config`: Specify configuration file.
- `-unique_world`: Whether or not to use unique URLs.
- `-local_rng_seed`: Seed for the random number generator.
- `-http_proxies`: Proxy server.
- `-verb_lvl`: Verbosity level.
- `-dump`: Items to dump.
- `-log`: Log file location.

By default, Polygraph runs its tests using a unique load. That means that the URLs are unique, and their request sequence is different from run to run. By setting the command line option `unique_world` to off, then the URLs generated will not be unique, making Polygraph free to use the same objects for various tests. To further ensure that the runs are identical for both proxy servers, Polygraph's random number generators should be provided the same seed, using the command line option `local_rng_seed`. While there is no guarantee that the random number generator will generate exactly the same numbers, nevertheless the results would be nearly the same[6]. The combination of these two options makes it possible to reproduce essentially the same URL sequence for separate tests.

To run the test, the server and client processes are run separately. They can be run on the same machine provided that they use different ports, or they can be run on different machines.

3.3. WEB POLYGRAPH

On the server side, the following command is used to start the Polygraph server for these tests:

```
server process
./polygraph-server --unique_world off --local_rng_seed 501 --config config.pg --verb_lvl 10 \
--dump errs --log results/log_server
```

On the client side, the following command is used to start the Polygraph client process:

```
client process
./polygraph-client --unique_world off --local_rng_seed 601 --config config.pg --verb_lvl 10 \
--dump errs --log results/log_client
```

3.3.3 Polygraph console output

The Polygraph console periodically reports the state of the test and the progress of its various phases. The phase duration (in minutes) and the fill size (in MB) as well as percentage completion of the current phase and the current frozen working set size are shown for each interval. Figure 3.4 shows an example of the client side console display.

```
CPU Usage: 13.76sec sys + 10.78sec user = 24.55sec
Maximum Resident Size: 213.531MB
Page faults with physical i/o: 6
1336627059.668174| fyi: phase progress:
duration:      5.00min
xact.count:    47604
fill.size:     220.65MB goal: 2.15GB (10.03% complete)
xact.errs.count: 11
xact.errs.ratio: 0.00
1336627059.668174| fyi: fill duration: 10.00min working set fill duration goal: 26.67min (37.50% complete)
1336627059.668174| fyi: min 'direct' objects in working set:
global public: 16049 (~140.80MB size, 0/1=0.00% frozen slices)
```

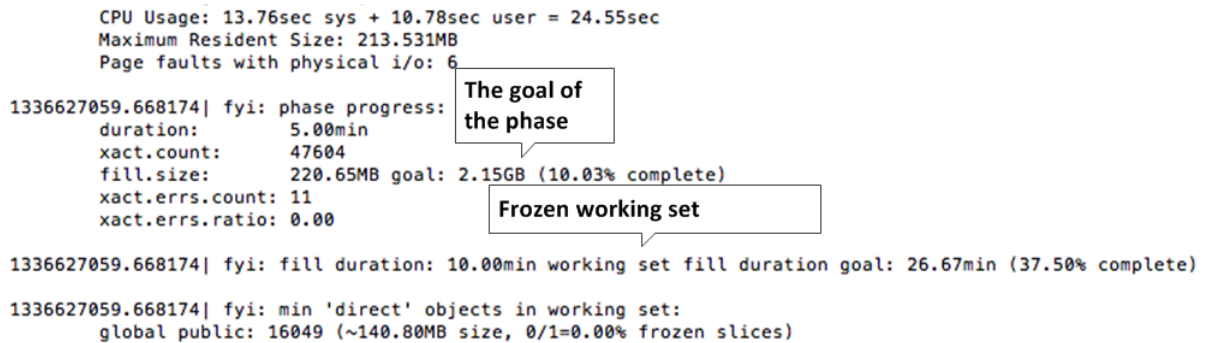


Figure 3.4: The current phase state: client side console output

Runtime messages are also displayed, at more frequent intervals than the preceding periodic status reports, in a tabular format. Any error messages are also included. Figure 3.5 identifies the most important parts of these messages.

3.4. SURROGATE TESTS

time	phase				Hit ratio		
1336627099.660176	i-fill	107061	159.40	294	1.74	0	489
1336627104.660177	i-fill	107906	169.00	298	2.98	0	519
1336627109.660219	i-fill	108768	172.40	295	1.86	0	503
1336627114.660176	i-fill	109627	171.80	296	2.81	0	490
1336627115.992509	HttpClxact.cc:844: error: 159/170 (c32) hit on reload request						
1336627119.660174	i-fill	110287	132.00	290	4.05	0	430
1336627124.660091	i-fill	111066	155.80	295	2.70	0	469
1336627129.660173	i-fill	111874	161.60	301	2.46	0	497
1336627134.660173	i-fill	112749	175.00	303	0.51	0	543

Figure 3.5: The runtime output: client side console

The Polygraph package also includes several additional tools along with the main servers. They have different functionality, including extract logs and troubleshoot runtime problems.

In addition to `polygraph-server` and `polygraph-client`, the `polygraph-reporter` command is used to generate a report from the server and client robots' log files. The generated report is in HTML format and can be viewed by any web browser. Here is an example command:

```
generating the report
./polygraph-reporter --label "results" --report_dir /var/www server.log client.log
```

Polygraph configuration and workload definition are discussed later in this chapter, in Sections 3.8 and 3.9, respectively.

3.4 Surrogate tests

Surrogate testing is originally a concept from medicine. When it is not possible to examine a body organ directly, so-called surrogate tests are performed that attempt to determine the characteristics of monitor the behaviour of that organ in a non direct manner. In the same way, various system metrics are monitor here in order to provide another view of proxy server functioning and performance. For this purpose, data for CPU usage, memory usage, and disk I/O are also collected to be considered in conjunction with the Polygraph results.

3.4.1 Blktrace

One of the aspects that will be monitored during the experiments is the disk usage of each accelerator. As is known, disk I/O can be a bottleneck for a web proxy when the read and writes operations become slow under heavy loads.

`blktrace` is a block layer I/O tracing tool which records detailed information about disk request queue events. The desired block device is specified with its `dev` option. `blktrace` records all events that happen on that device as long as it

3.4. SURROGATE TESTS

is running, storing the generated trace to the file specified to the `o` option[63].

`blktrace` generates a separate file for each CPU. By default, when an output file name is given, the generated event trace is stored as `filename.blktrace.0` and `filename.blktrace.1` for a two core system.

The following is an example `blktrace` command:

```
blktrace run  
  
blktrace -d /dev/sda1 -o output -n 4 -b 4096
```

The options used are:

- `-d`: The device name
- `-o`: The output file
- `-n`: The number of buffers which store the collected data
- `-b`: The size of buffer(s)

Since the test duration may be quite long, the size of the `blktrace` output file may easily become very large, consuming several gigabytes. In order to prevent a disk resource shortage during the Polygraph run, a customized script was written to start `blktrace` only after the test has reached to the link phase. This serves to eliminate the events generated during the fill phase, which are not important and need not be recorded, saving a considerable amount of disk space. The script can be found in the Appendix.

3.4.2 Seekwatcher

Seekwatcher is a tool which can generate simple graphs from the traces made by `blktrace`. It utilizes the `matplotlib` library to visualize the I/O patterns and performance recorded by `blktrace`. It is possible to create graphs from with multiple traces in order to easily compare the differences between separate benchmark runs.[64] In order to do so, the option `-t` is used multiple times with the `seekwatcher` command:

```
seekwatcher can make graphs for multiple traces  
  
seekwatcher -t varnish.blktrace.0 -t trafficserver.blktrace.0 -l Varnish -l ATS \  
-o varnish_vs_trafficserver.png
```

The three command line parameters specify the following:

- -t: the blktrace trace file
- -l: assigns a label to the specified trace in the output graph
- -o: specifies the name of output file

3.5 Varnish Configuration

Varnish uses its own configuration language, called VCL (Varnish Configuration Language). There are two main files in the configuration. In the `/etc/defaults/Varnish` file, a number of runtime elements for the Varnishd daemon are configured.

- -a :80 – The address:port that Varnish is listening to. Here, Varnish is listening to localhost and the port http(80).
- -T 127.0.0.1:6082 – The management interface address. Using this IP address/port pair, which in this case is the localhost and port 6082, it is possible to manage the Varnishadm daemon in real time without stopping the daemon.
- -P `/var/run/Varnish.pid` – The file storing the running Varnish parent PID.
- -S `/etc/Varnish/secret` – The secret file used for authentication, required for the Varnishadm utility.
- -t 0 – The hard time to live or ttl for objects which are cached/ Here, it is set to zero.
- -f `/etc/Varnish/default.vcl` – Varnish is informed of the location of the VCL file by this option (discussed below).
- -s `file,/var/lib/Varnish/$INSTANCE/Varnish_ storage.bin,1G` – Varnish supports three types of storage. On disk, memory and a persistent mode. Here, Varnish is configured to use the given file on disk with the size of 1 GB.

In the `/etc/Varnish/default.vcl` VCL file, the vcl methods `vcl_recv`, `vcl_fetch`, `vcl_pipe`, `vcl_pass`, `vcl_hash`, `vcl_hit`, `vcl_miss`, `vcl_deliver` and `vcl_error` are defined. The backend(s) are also set here. In this case, the experiment's web server is specified.

```
backend polygraphServer {
    .host = "192.168.0.2";
    .port = "9000";
}
```

3.6 Traffic Server Configuration

Apache Traffic Server uses different configuration files. Since ATS is initially a forward proxy, the reverse proxy mode must be activated. The main configuration file is `/etc/trafficserver/records.config`.

Those parts that have been edited from the default are shown below. In line 1, ATS is instructed to listen to port 80. The second line is the amount of RAM the proxy should use (here 1GB). In line 3, the reverse proxy mode is enabled. In the final line, the server redirects requests without any host address to the machine 192.168.0.2 port 9000.

```
CONFIG proxy.config.http.server_port INT 80
CONFIG proxy.config.cache.ram_cache.size INT 1073741824 # 1G

CONFIG proxy.config.reverse_proxy.enabled INT 1
CONFIG proxy.config.header.parse.no_host_url_redirect STRING http://192.168.0.2:9000
```

Next, the server should be assigned to the back end server. The map key in the `/etc/trafficserver/remap.conf` file tells ATS to look for the URLs beginning with the first field one the back end server specified in the second field:

```
map http://192.168.0.2:9000/ http://192.168.0.2:9000/
```

In the file `/etc/trafficserver/storage.conf`, the server is instructed to use 1 GB of disk, located at the specified location:

```
/var/cache/trafficserver 1073741824
```

3.7 The custom scripts

When running each test, a number customized scripts are used to simultaneously monitor the resource usage. The results should be extracted from the log files, and a number of other tasks should be done. The main script does the task, and organizes the results in a structured way that is easier to reach the results. The following pseudo code outlines the structure of the script:

```
Automation script: polystart.pl

<create log files>
<refresh the system caches>
<refresh the proxy cache>

while (exist proxies) {
  <start the proxy>
  <start the server and client processes>
  while (<client process runs>) {
```

3.7. THE CUSTOM SCRIPTS

```
        if (<reached the link phase>) {
            <run the blktrace script>
            <run the vmstat script>
            <run the top script>
        }
    } # end client process run
} # end loop over proxies

<stop all other scripts>
<run seekwatcher>
<copy files from all machine to result directory>
<generate the polygraph report>
```

The polystart.pl scripts calls another script(awkp.pl) on the client machine in order to determine whether the client processes are running and what phase they are currently in. Based on its output, polystart.pl decides whether to quit.

```
----- awkp.pl -----
#!/usr/bin/perl
my $num= "ps aux | grep polygraph | wc -l ";
system("$num");
```

The blktrace.pl script controls the running of blktrace. The following pseudo code illustrates its structure:

```
----- blktrace.pl -----
<get the proxy name>
<get the output file name>

\ $command = "blktrace -d \ $dev -b 4096 -n 20 -o \ $output";
<run \ $command>
```

The top.pl script records key system resource metrics while the experiment runs. Here is its structure in pseudo code format:

```
----- top.pl -----
<get the proxy name>
<get the output file name>

open(output file)

while (polygraph-client runs) {
    find_the_proxy_PID();
    <extract the top command entry for PID>
    <extract the MEMORY and CPU PERCENTAGE>
    <write TIME, MEMORY CPU\& to output file>
}

<close output file>
```

3.8 Polygraph Configuration

Polygraph can freely be configured using PGL in configuration files. The configuration file then is provided to the running server and client processes.

The following pseudo code describes the general organization of the Polygraph configuration file:

```

pseudo polygraph configuration file
<include contents.pg>
<include phases.pg>

calculate { <Peak rate, Fill rate, Working Set Size, Cache Size, ... >}

IP Addresses { <client, server, proxy> }

Server S = { <server agent configuration> };

Robot R = {<robot configuration> };

PeakRate = (number of robots) x (robot.req rate)

phases {<define different phases>
        <set various parameters>
      }

schedule( <sequence of the phases in the run time> );

calculate working set length>

address assignment <client><server>

<commit configuration of these servers and robots>

```

3.8.1 IP Addresses

After configuring the network metrics, the Polygraph servers and clients are configured. First, the number of robot (client) agents is specified for the workload. When the Polygraph process starts, the agent finds the network interfaces on the system and associates the client and server addresses with the interface. The "cloning" feature of PGL (denoted by ** in the configuration file) allows Polygraph to clone the interface and assign as many as robots to each interface as is desired. See lines 38-40 in the configuration file attached in the appendix [6.2](#).

```

IP addresses
addr[] srv_ips = ['192.168.0.2:9000'];
addr[] rbt_ips = ['192.168.0.3' ** 1000];
addr[] proxy_ip = ['192.168.0.1:80'];

```

The above assignments create 100 client agents associated with the eth1 interface on the IP address 192.168.0.3. The server robots are instructed to listen to the IP address 192.168.0.2:9000. The client robots talk to proxy on port 80,

3.8. POLYGRAPH CONFIGURATION

and the proxy is configured to fetch the objects corresponding to cache misses from the backend server (process) at 192.168.0.2:9000.

3.8.2 Working Set Size

The working set size (WSS) is an amount of memory chosen to correspond to the set of objects that have a non-zero probability to be requested at any given time of test. This variable mimics the size of the simulated web site whose traffic is being accelerated. A medium-sized web site which is worth accelerating begins with around 100 MB of data at the start of the Polygraph simulations. A 1 GB working set size is used for large websites. The working set size is an initial value which is gradually increased as needed during the course of the Polygraph run.

The WSS could be any arbitrary number, but it should not be much smaller than the cache to get realistic results. Otherwise, the entire working set can be cached in the memory. On the other hand, if the WSS is much larger than the cache size, Polygraph may generate a lot of new objects before freezing its size, resulting in a very low hit rate, which is again unrealistic.

size WSS = 100 MB;

A WSS of 100MB is used in several tests in this research, but it is not the standard size for all. Each workload/scenario uses own working size set. To calculate an adequate initial WSS for each test, the size of the cache is divided by mean object size for that test. However, the benchmarking process should introduce new cachable objects into the working set during the course of the run to simulate cache misses. This causes the WSS to grow. The problem comes when the working set size grows so large it exceeds the system's memory capacity or causes the hit rate to drop to an unacceptable level.

The solution is to limit (freeze) the size of working set. The size should be small enough to not take an inordinately long time to reach the steady state but large enough to make the proxy to cache a realistic amount of cachable objects. Polygraph solves this problem by defining `working_size_length` which is the WSS's time rather than number of objects. See line 78 in configuration file.

working_set_length(WSS/(10KB*FillRate)/5%/80%);

This configuration line means that the working set can grow until the time that the fill phase has reached 80% of it phase time. When running the test, the time is given in minutes. It is dependant on the fill rate, and this it is defined as an expression rather than as a fixed value.

3.8.3 Cache size

The size of the cache should be set to the configured size of the disk cache set for the proxy server. In addition, since the proxy uses part of the memory to index the objects, as well as to store very popular objects, the total amount of memory used by the cache also should be added to the preceding when specifying the cache size attribute.

Apache Traffic Server and Varnish are configured to use 100 MB of RAM, and the total cache size configured for the accelerator is 1 GB. Since Varnish and Traffic Server do not use the same amount of the memory, it is not possible to select one value that is correct for both. Thus, the cache size is set to the disk cache size, and the cache utilization is measured by the surrogate tests during the experiment.

```
size CacheSize = 1100MB;
```

Object Life Cycle

The Object Life Cycle (olc) component of Polygraph is in charge of the modification and expiration of the objects. All the objects have a cyclic life cycle, meaning that the objects are modified periodically. The modification of objects is mostly independent and differs from object to object. The changes are not done at constant intervals but rather may vary over time. The variance variable in olc objects gives Polygraph the possibility of modifying the objects with realistic behaviour.

```
// HTML object life cycle
ObjLifeCycle olcHTML = {
    length = logn(7day, 1day);
    variance = 33%;
    with_lmt = 100%;
    expires = [nmt + const(0sec)];
};

// Image object life cycle
ObjLifeCycle olcImage = {
    length = logn(30day, 7day);
    variance = 50%;
    with_lmt = 100%;
    expires = [nmt + const(0sec)];
};

// Download object life cycle
ObjLifeCycle olcDownload = {
    length = logn(0.5year, 30day);
    variance = 33%;
    with_lmt = 100%;
    expires = [nmt + const(0sec)];
};

// object life cycle for "Other" content
ObjLifeCycle olcOther = {
    length = unif(1day, 1year);
};
```

3.8. POLYGRAPH CONFIGURATION

```
    variance = 50%;
    with_lmt = 100%;
    expires = [nmt + const(0sec)];
};
```

The Object Life Cycle olcHTML specification has four variables which will be explained below.

- **Length:** The cycle duration or lifetime specified as a distribution function. For example, `logn(30day,7day)` is a heavy tailed (logarithmic) model, which updates each month with deviation of 7 days
- **variance:** The percentage of modification time variation from the middle of the interval.
- **with_lmt:** The percentage of objects that include the last modified time header from the server side. For example, 100% means that this header is present in all the objects.
- **expires:** Specifies when objects expire (as an expression). For example, the `nmt + const(0sec)` value means that at time of next modification, the object will be expired.

3.8.4 Content types

Polygraph can generate objects having a variety of content types. In the contents part of the configuration, the server agents are told what kinds and percentages of the various content type are going to be used. Supported content types for polygraph are Image, HTML, DownloadD and Other, which have the following specifications:

```
Content cntImage = {
    kind="image";
    mime={type=undef(); extensions=[".gif", ".jpeg", ".png"]};
    obj_life_cycle=olcImage;
    size=exp(4.5KB);
    cachable=80%;
    checksum=1%;
};

Content cntHTML = {
    kind="HTML";
    mime={type=undef(); extensions=[".html" : 60%, ".htm"]};
    obj_life_cycle=olcHTML;
    size=exp(8.5KB);
    cachable=90%;
    checksum=1%;
    may_contain=[cntImage];
    embedded_obj_cnt=zipf(13);
};

Content cntDownload = {
    kind="download";
    mime={type=undef(); extensions=[".exe":40%, ".zip", ".gz"]};
```

3.8. POLYGRAPH CONFIGURATION

```
obj_life_cycle = olcDownload;
size=logn(300KB, 300KB);
cachable=95%;
checksum=0.01%;
};
```

The fields `kind` and `mime`, as their names suggest, indicate the type name and allowed file name extensions of the associated objects. The `size` is the range of sizes for the objects, and it is defined by three different models: the logarithmic model, `logn()`, for long tail distributions, the exponential model, `exp()`, for object sets where the size may vary highly, and the uniform, `unif()`, model. The parameter to the model function specifies the mean value for the distribution.

The `cachable` variable determines what many percent of this type object should be cachable. The `checksum` variable determines the percentage of objects which have a MD5 checksum, which is computed and attached to the Content-MD5 header.

3.8.5 The phases

The following phases are used during the tests' run time. Phases are accomplished one after another, after each phase reaches its goal. The following are definitions of several of the phases:

```
Phase phWarm = {
    name = "warm";
    goal.duration = 5min;
    load_factor_beg = 0.1;
    load_factor_end = FillRate/PeakRate;
    log_stats = false;
};

Phase phFill = {
    name = "fill";
    goal.fill_size = 2*CacheSize;
    recur_factor_beg = 5%/95%;
};

Phase phLink = {
    name = "link";
    goal.duration = 5min;
    load_factor_end = 1.0;
    recur_factor_end = 1.0;
};

Phase phTop1={name="top1";goal.duration=60min;};

Phase phDec={name="dec"; goal.duration=5min;load_factor_end=0.1;};

Phase phIdle={name="idle";goal.duration=10min;};

Phase phInc ={name="inc";goal.duration=5min;load_factor_end=1.0;};

Phase phTop2={name="top2";goal.duration=120min;};
```

3.8.6 Server configuration

The server configuration specifies the content type selection and distribution for the run. The content types specified to the contents variable can directly be requested from the robots.

In the example below, the server latency, or think time (`xact_think`), is configured to follow a normal distribution with a mean of 300 milliseconds. The number of requests in the persistent connections follows a Zipf distribution with the mean 16 (`pconn_use_lmt`). The timeout for idle connections is set to 15 seconds to free the ports and make better use of resources (`idle_pconn_tout`).

```
kind = "WebAxe-1-srv";
contents = [ cntImage: 65%, cntHTML: 15%, cntDownload: 0.5%, cntOther ];
direct_access = [ cntImage, cntHTML, cntDownload, cntOther ];
xact_think = norm(0.3sec, 0.1sec);
pconn_use_lmt = zipf(16);
idle_pconn_tout = 15sec;
http_versions = [ "1.1" ];
```

3.8.7 Client configuration

This is the main configuration part that provides instruction to the robots that will generate the traffic for the run. The proxy address is given by the `http_proxies` variable. The recurrence field is the probability that a previously requested object will be asked for again. In this case, 95% of pages will be requested again. The `embed_recur` variable says that all (100%) of the embedded objects should be requested via a persistent connection. The interests field specifies the percentage of common URLs that robots can use versus the URLs specifically for this robot designed.

```
kind = "WebAxe-1-rbt";
origins = srv_ips;
http_proxies = proxy_ip;
recurrence = 95%;
embed_recur = 100%;
interests = [ "public": 90%, "private" ];
pop_model = { pop_distr = popUnif(); };
req_rate = 1/sec;
pconn_use_lmt = zipf(64);
open_conn_lmt = 4;
http_versions = [ "1.0" ];
```

When clients want to generate hits, they use the `uniform()` distribution for the popularity model of object set. As discussed in the section [2.6.1](#), according to Zipf's law, when the time factor is shortened, the hot data set will become too small and will be mostly served from the memory rather than disk for a simulated environment. Therefore, the uniform model is used.

The `uniform()` model initially ranks all the objects with the same probability, meaning that it chooses the object randomly from the the set of objects which previously have been requested. However, since the objects from the beginning of the test are more often requested than the later objects, it still implies some notion of popularity.

3.9 Defining the Experimental Phases and Workloads

As noted above, one key requirement for an experiment to run successfully is that the `frozen working set` should be sized appropriately. The frozen working set is the maximum size to which the original working set which grows during the fill phase to generate cache misses. However, this parameter cannot be set directly. Moreover, the growth of the working set varies according to several factors, including the length of the fill phase, the object size range and type, the extent of object cache-ability, and the like. Thus the size of frozen working set is different from workload to workload despite using same WSS setting of 100 MB. Thus, substantial experimentation and tweaking is required to determine the proper parameters for a workload in order to achieve the desired traffic characteristics and other properties.

Several different workloads were designed and tuned for use in this research. They are described individually in the following subsections.

3.9.1 The best effort method

In this scenario, the robots were configured to use the best effort method. The best effort method means that a robot sends the next request as soon as a response is received for the previous request, but not sooner. In practice, this means that no robot makes parallel connections at any given time. In this scenario, the cache never gets overloaded. This method shows how fast a proxy is by "nature" – what request rate the proxy is convenient with – and the rate of requests is indirectly justified by the proxy itself.

To run such a test, the `req_rate` variable of robot objects is set to null or is simply removed from the robot part of the configuration files. The workload specifications which were used in this part are given in Table 3.3. Note that the frozen working set size is the one resulting from specifying the other parameters.

3.9. DEFINING THE EXPERIMENTAL PHASES AND WORKLOADS

Best Effort workload		
	Varnish	Apache Traffic Server
Working set size (MB)	100	100
Cache Size (MB)	1100	1100
Frozen WSS (GB)	2.9	3.2
Mean object size (xact)	147625	141027
Robot population	100	100
Content Type	All	All
Peak Rate	not specified	not specified

Table 3.3: Parameters for the Best Effort workload

The following graph illustrates the content type mix for this workload.

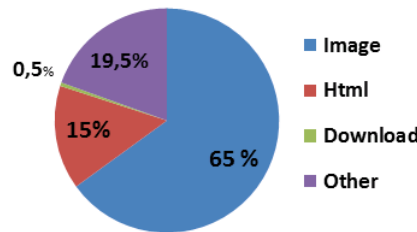


Figure 3.6: Content type distribution

The phases used for this workload are described in Table 3.4.

Phase	Request rate(%Peak)	Recurrence (%)	Goal
warm	10 to 100	5	5 min
fill	100	5	2*Cache Size
link	fill to 100	95	5 min
top1	100	95	60 min
dec	100 to 10	95	5 min
idle	10	95	10 min
inc	10 to 100	95	5 min
top2	best effort	95	120 min
cool	100 to 0	95	1 min

Table 3.4: The Phases for the Best Effort Workload

3.9.2 Baseline Workloads

A series of baseline workloads were created in order to measure cache performance under controlled circumstances with limited amounts of variation. These workloads use a single content type and the constant method request rate profile. In the constant method, the robots are instructed to send requests at a constant rate throughout the test. The `req_rate` variable for each robot specifies the interval after which the client should send a new request.

3.9. DEFINING THE EXPERIMENTAL PHASES AND WORKLOADS

req_rate = 0.4/sec;

In these workloads, each robot is instructed to issues requests at an average request rate of 0.4 requests/sec. While the clients are capable of generating much higher request rates, that is somewhat artificial in that typical real world clients in the web do not generate such large amounts of requests. Therefore a small value for each individual client was chosen.

A large part of this work was conducted using individual types of objects. That means that the three major type of web objects (Images, HTML and Download) are used one by one to determine the baseline behaviour for each proxy. The workloads cannot be totally identical in these cases. For example, using a working set size which is suitable for HTML objects would generate a much higher, excessive frozen working set for Downloads and a smaller frozen WSS for Images. Thus, each workload has its own specifications. The specifications used for each workload are specified in the corresponding section of the results discussion in Chapter 4. However, the phases are identical for each baseline and are shown in Table 3.5.

Phase	Request rate(%Peak)	Recurrence (%)	Goal
warm	10 to 100	5	5 min
fill	100	5	2*Cache Size
link	fill to 100	95	5 min
top1	100	95	60 min
dec	100 to 10	95	5 min
idle	10	95	10 min
inc	10 to 100	95	5 min
top2	100	95	60 min
cool	100 to 0	95	1 min

Table 3.5: The Phases for each individual content type workload

3.9.3 The increasing rate workload

After completing the baseline experiments, a final scenario were designed to combine all the object types. In this case, the request rate was gradually increased during one phase in order to simulate a real, stressful load for the cache. The phases and workload specifications for this workload are described in conjunction with the results in Chapter 4 in order to facilitate the discussion. The results for both `top1` and `inc` phases are important in this test. The `top1` phase is a phase with a relatively low request rate which is not stressful and simulates a normal web traffic situation. In the phase `inc`, on the other hand, the load is increased gradually to reach to the saturation points of the proxies.

Chapter 4

Results and Analysis

This chapter will discuss the results of the Polygraph tests for the two cache servers described in the previous chapter. They consist of the following:

- Sample output from a complete Polygraph run showing all phases.
- Results from the best effort workload to illustrate the proxy servers' natural throughput and processing rates.
- Single content-type workload results: Image-only, HTML-only and download-only workload results will be examined in order to establish baseline performance levels for various types of web content.
- Increasing rate mixed workload results. There are two distinct parts to this test. In the first part, Polygraph's top1 phase, there is a steady request rate in order to test both cache servers under completely identical workloads and conditions. In the second part, the request rate increases gradually to 100% of the peak rate in an attempt to saturate the cache server. Both phases use a mix of content types for testing.

4.1 Sample Polygraph results

Figure 4.1 below illustrates the general form of Polygraph result plots. The time in minutes forms the X-axis with the measured results plotted as Y (here, the request rate per second). The various phases of the Polygraph run are labeled (as defined previously in Figure 3.3).

4.1. SAMPLE POLYGRAPH RESULTS

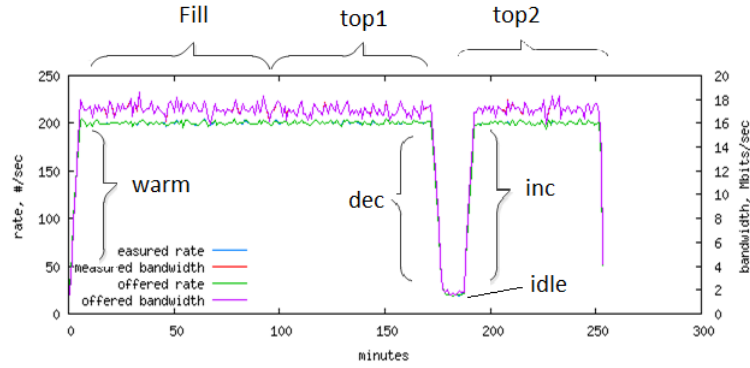


Figure 4.1: The General form of Polygraph result plots

For each run, Polygraph produces many graphs of this type for all of the various data it collects. In general, the early periods of the test have a different data pattern from the remainder. Thus, in many cases, only the rightmost parts of the graphs are important for analysis. The comparisons in this chapter are mainly conducted between the later phases: top2 for the best effort workload and baseline comparison, and top1 and inc for the mixed workload. This is the period where the proxies are in their steady state and give the most valid and meaningful results.

Polygraph produces several different kinds of plots. Many of them are analogous to the one above, plotting the value of a collected data items or comparing two related ones, such as offered vs. achieved (measured) bandwidths. Another type of plot is the following:

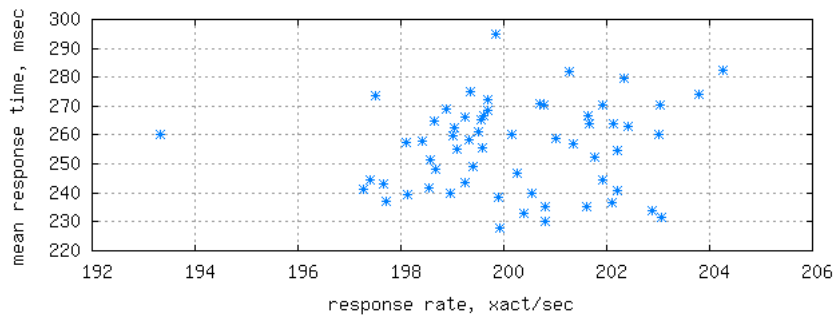


Figure 4.2: The of Polygraph result scatter plots

Figure 4.1 is a scatter plot of the response rate (X) vs. the mean response time

4.2. BEST EFFORT WORKLOAD RESULTS

for requests at that rate (Y). In this particular example, the data is widely divergent and shows no particular pattern.

In the sections that follow, the separate Polygraph results for Varnish and Apache Traffic Server (ATS) are combined into a single plot using the gnuplot application [65].

4.2 Best effort workload results

In this section, the results from the best effort workload are presented. In the best effort scenario, the next request is made as soon as the response to the previous request is received. This ensure that the cache does not get overloaded. Accordingly, it reveals the "natural" request rate that the proxy supports/is comfortable with.

Figure 4.3 illustrates the request rate (xact/sec) and bandwidth usage (Mbit-s/sec) during the experiment. The results show that ATS managed to handle a request rate with a mean of 520.6 xact/sec (the green line), achieving a throughput of 49.6 Mbits/sec (the pink line), while the results for the Varnish server are 500.3 xact/sec and 43.4 Mbits/sec (the green and red lines, respectively).

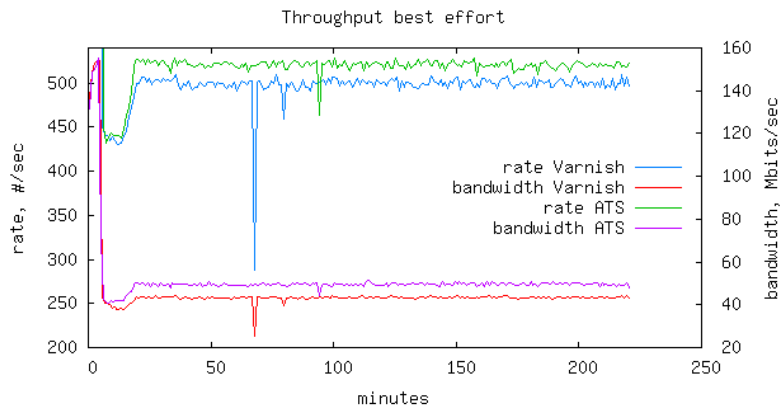


Figure 4.3: Best effort request rate and throughput

Figure 4.4 shows the response time for each server. As the chart indicates, the response time for cache hits is considerably lower with Varnish than for ATS. In contrast, the two servers' the response time plots for misses are virtually the same line, at around 300 ms.

4.2. BEST EFFORT WORKLOAD RESULTS

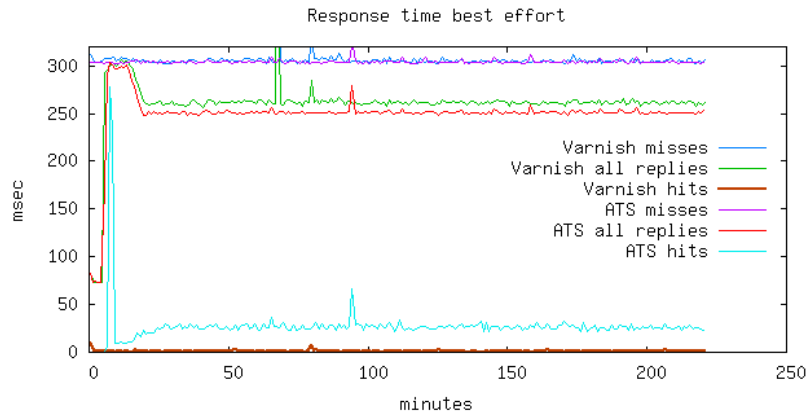


Figure 4.4: Best effort response times

The mean response time is slightly better for ATS at 251 ms, compared to 261 ms for Varnish (a difference of about 4%).

Figure 4.5 compares the offered and measured document and byte hit rates for both servers. The Y axis plots the percentage of cacheable documents or bytes for all the requests over the time plotted on the X axis. With the offered rate of 76% of the peak rate, Varnish kept 17.2% of the documents, which corresponded to 16.6% of the transferred bytes. Traffic Server cached 19.0% of the documents, which was 19.1% of the offered data in bytes.

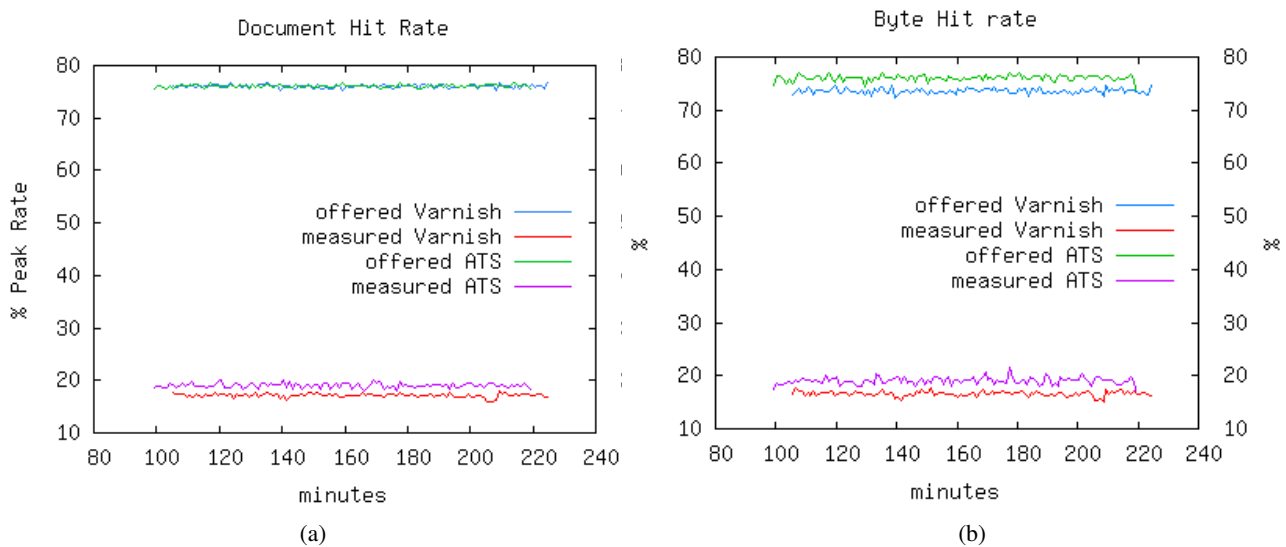


Figure 4.5: Best effort document and byte hit rates

Figure 4.6 show the distribution of response rates by response time. The X axis shows the response rate as seen from the client side. The Y axis shows the corresponding mean response time for that specific response rate. ATS values

4.2. BEST EFFORT WORKLOAD RESULTS

are plotted in red, and they are mostly concentrated between 515 and 525 milliseconds, with a mean response rate of 520.6. For Varnish (in blue), response times are clustered between 248 and 254 msec, with a mean of 251. In this case, the performance of the Varnish server exhibits lower response rates and higher response times.

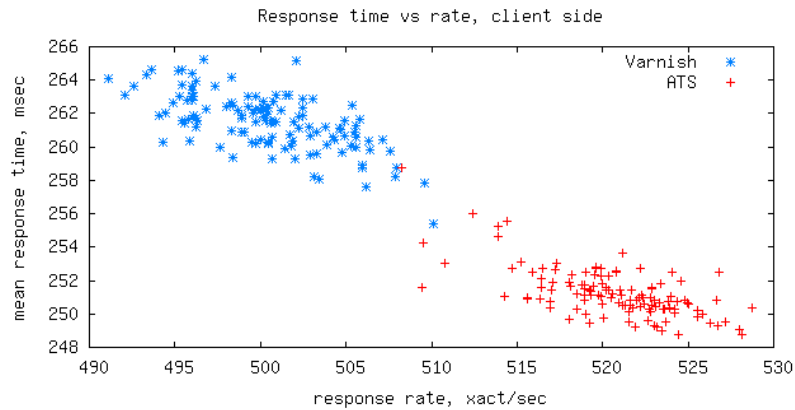


Figure 4.6: Best effort response time vs request rate distribution, client side

Seen from the server side, the situation is quite different. In Figure 4.7, the response time is quite similar for both servers, but the response rate is higher for Varnish. Note that the spread of response times is much narrower here (about 4 msec vs. 28 msec from the client side), although the magnitude is higher by about 40 msec; the response rate range is also different, by about 100 transactions/sec in favor of the client side. These observations can be explained by noting that the traffic on the server side is much lower than the originating traffic on the client side due to the effects of the proxy server.

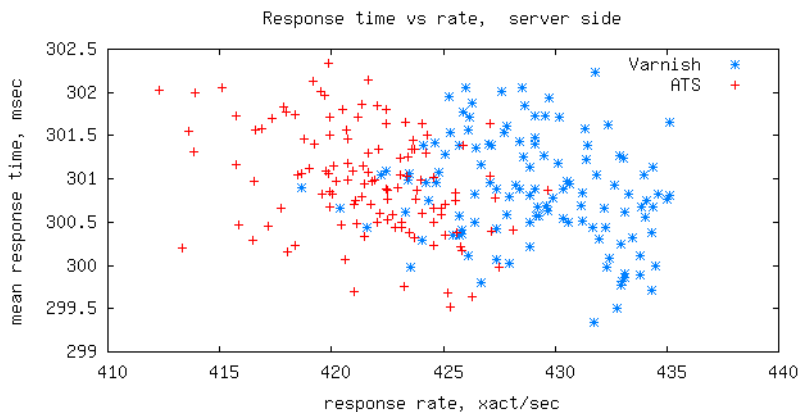


Figure 4.7: Best effort response time vs request rate distribution, server side

4.3 Single content type workload results

The following sections discuss the results of Polygraph experiments for single types of web items. The purpose of these test is to benchmark proxies with only a single type of content at a time.

Some care was required in order to ensure that the workloads created by Polygraph were the same for the two servers and also remained within realistic resource limits. The desired cache capacity was decided to be 1GB. Polygraph’s frozen working set size, which represents the total website content, should be very close to this size. However, this parameter cannot be set directly but rather is computed by Polygraph based on the initial working set size and the content object size distribution and mean.

Selecting values for these parameters designed to mimic real world web usage as closely as possible did not always result in reasonable workloads. For example, using a starting working set size of 100MB and an image mean size of 4KB (exponential distribution), resulted in a frozen WSS of only 400-600MB. Even worse, using the same 100MB initial working set and a 300KB object size for the mean download object size (logarithmic distribution) resulting in a frozen WSS of 33GB! Experimentation ultimately lead to the values used in the final tests presented here. These parameters are summarized for each content type at the beginning of their discussion.

4.3.1 Results for image content type

Table 4.1 gives an overview of the configuration variables for the image content type scenario. Table indicates that the 1000 clients generate a load of 400 requests/sec using a starting working size set of 100 MB. The workload parameters specify an exponential distribution for the images sizes with a mean of 9KB. This results in a mean size of objects generated of 4612.66 bytes for the constructed workload, as well as a working set freeze size of about 640MB. The phases for this workload were described earlier in Table 3.5.

The image workload		
	Varnish	Apache Traffic Server
Working set size (MB)	100	100
Cache Size (MB)	1000	1000
Frozen WSS (MB)	640	640
Mean object size (bytes)	4612.66	4612.66
Robot population	1000	1000
Content Type	Image	Image
Peak Rate (req/s)	400	400

Table 4.1: Image workload defined and generated parameters

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

This workload consists of 3 major phases, including with 2 top phases and one idle phase. In the idle phase, which is a low rate of 10% of the peak rate (40 xact/sec), both Varnish and Traffic Server utilize the full offered rate, corresponding to a bandwidth which of 2MB/sec. In the top2 phase, Varnish and Traffic Server perform almost the same, with mean rates 400.11 vs 400.05 xact/sec and mean bandwidths of 15.3 vs 15.34 MB/sec (respectively).

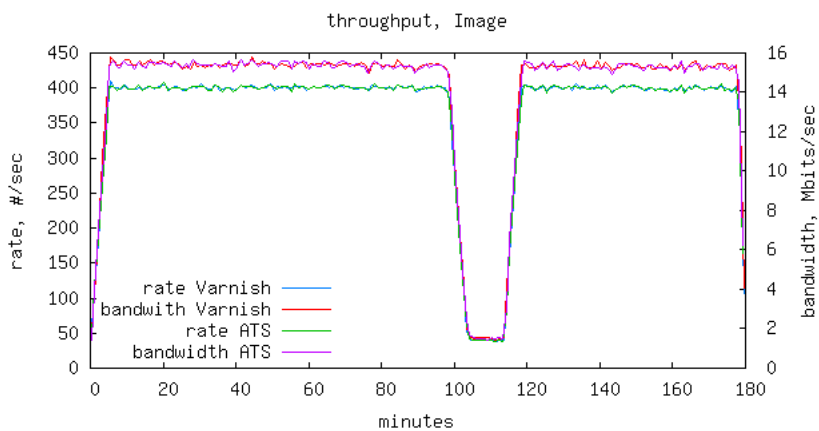


Figure 4.8: Image content type throughput

When it comes to response time, Figure 4.9 illustrates that the response time in this phase is higher for Varnish. The mean time for all responses is 78 msec for ATS (the brown line), while the response time for Varnish is 122 msec (the light blue).

The response time for misses is almost the same for either case with a few small spikes for ATS. When it comes to hits, The response time is better for Varnish with just a few msec (1-3 msce) while the time for ATS is longer with occasional spikes.

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

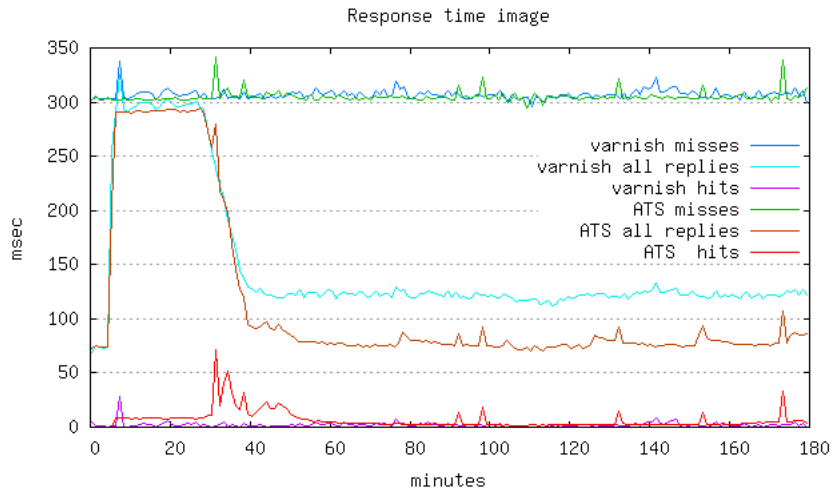


Figure 4.9: Image content type response time trace

The charts 4.10a and 4.10b in Figures 4.10 show the document and byte hit rates. The green line shows the offered percentage of cachable data, whose mean 76% in both cases. The pink line shows Traffic Server’s performance: a document hit rate (DHR) of 75.1% and a byte hit rate (BHR) of 75.5% for all transactions. In comparison, the DHR and BHR for Varnish are 60.5% and 60.9%, respectively.

ATS hit rate is higher than Varnish(the purple vs red line). However we can see a periodic fall of 2% whenever it reaches to the highest level (the green line). Varnish hit rate (red line) seems more stable with an almost level line during the whole phase.

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

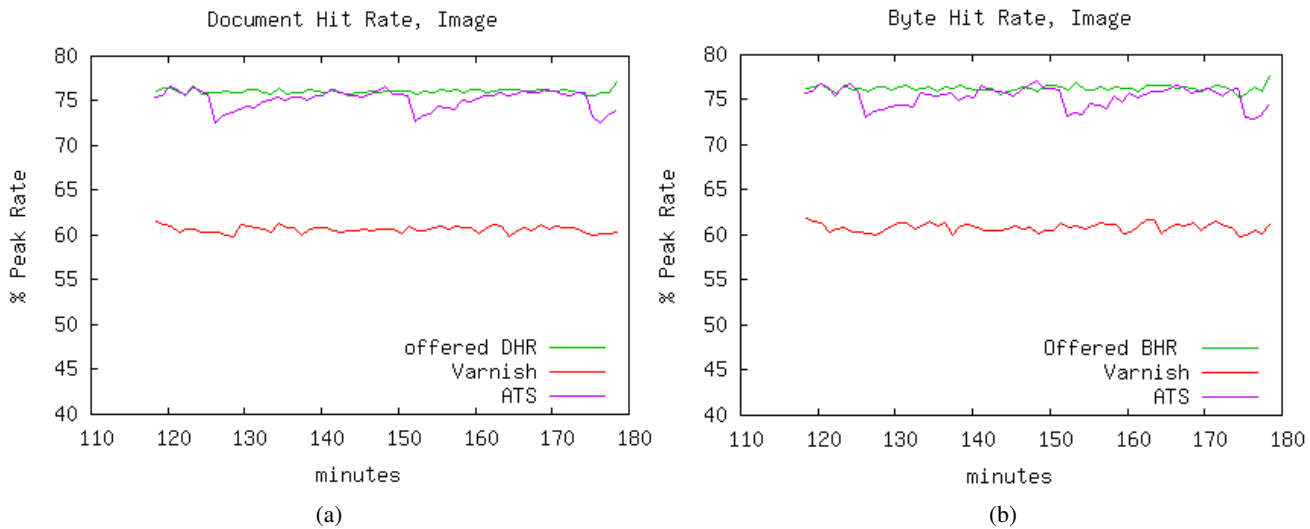


Figure 4.10: Image content type document and byte hit rates

Figures 4.11a and 4.11b show the response rate vs response time rate distributions for the client and server sides, respectively. The first figure shows that the Varnish server responded to the request rate of 400 requests/sec (with 5 request deviation) with a constant mean response time of 120-130 msec. ATS responded to the same workload with response rates ranging from 70 to 90 requests/sec.

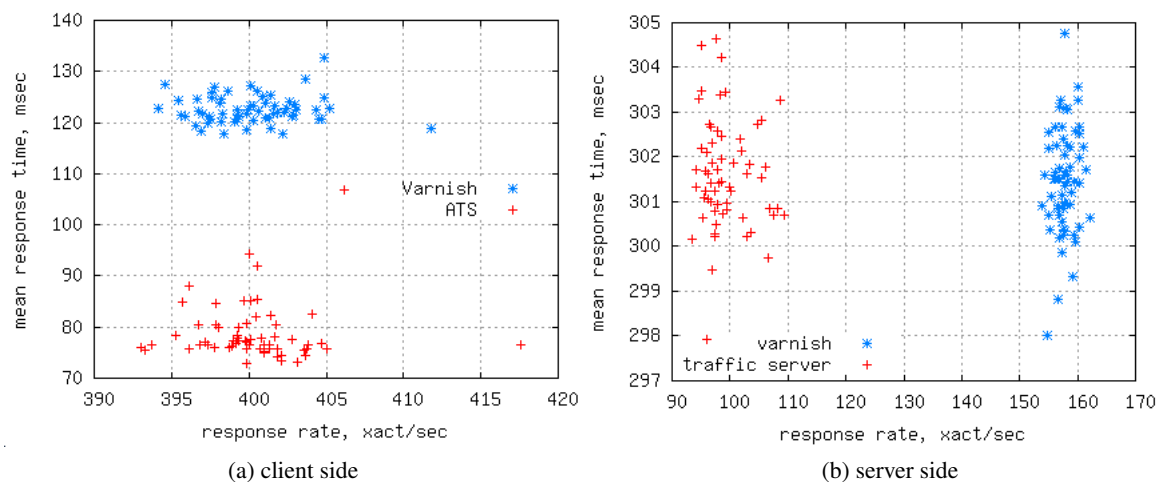


Figure 4.11: Image content type response rate vs response time

On the server side, the difference is much less when it comes to the response time. Varnish and ATS both response with a mean response rate between 298 and 304 xcat/sec. The response rate for ATS is 90-110 requests/sec while it is 150-160 xect/sec for Varnish. This illustrates a clearly higher response rate

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

for Varnish with almost the same response time with ATS.

4.3.2 HTML content type results

Table 4.2 shows the workload specifications used for HTML object type benchmarking. The phases are as described previously in Table 3.5.

The HTML workload		
	Varnish	Apache Traffic Server
Working set size (MB)	100	100
Cache Size (GB)	1	1
Frozen WSS (GB)	1.12	1.14
Mean object size (bytes)	8713.25	8713.25
Robot population	1000	1000
Content Type	HTML	HTML
Peak Rate (req/sec)	400	400

Table 4.2: HTML workload defined and generated parameters

Figure 4.12 shows the throughput for the HTML content type. The mean transaction rate for Varnish is 400.2 requests/sec, and the mean bandwidth usage is 27.4 MB/sec. The mean rate for ATS is a bit lower: 324.79 xcat/sec and 22.65 MB/s bandwidth usage. Thus, Varnish throughput is higher and the load utilization is more stable, as it is seen in the larger rate fluctuations in the graph for ATS (purple line).

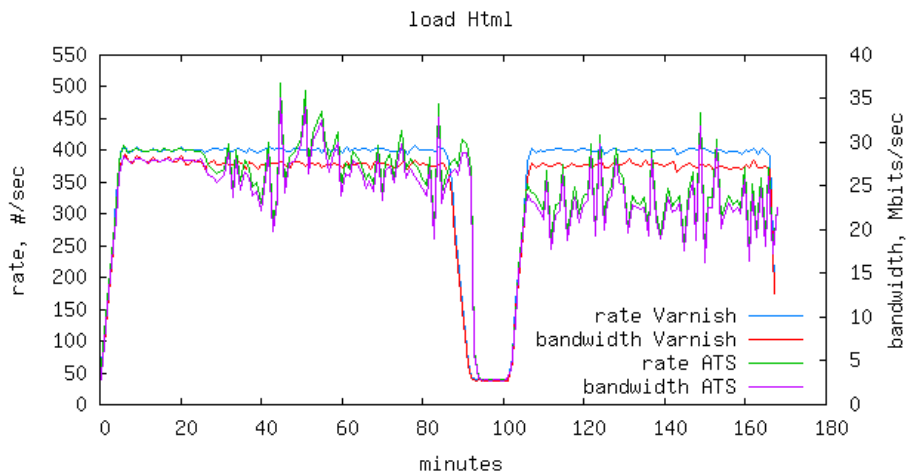


Figure 4.12: HTML content type throughput

Since the response time difference is very high, and it is difficult to scale the data for both servers in a single picture, they are shown in separate figures below. Figures 4.13 and 4.14 show the response time for Varnish and ATS, respectively. The mean response time for Varnish cache hits is 1 msec, while the corresponding value for cache misses is 300 msec for misses, resulting in an overall mean of 152 msec for all responses. In contrast, Figure 4.14 shows the

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

response times for ATS, which are all 2 orders of magnitude higher. As with the image workload, the response time is higher for hits than for misses for ATS.

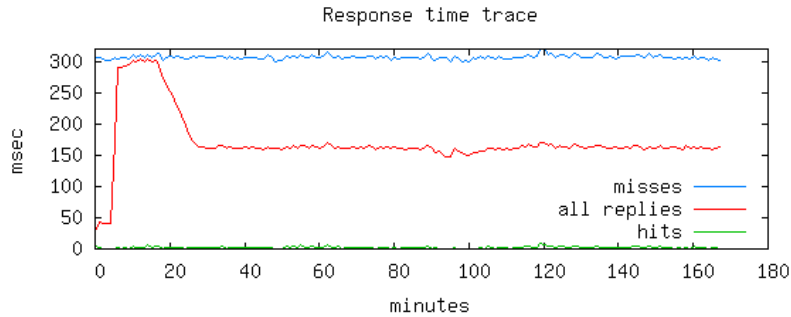


Figure 4.13: Response time trace, Varnish

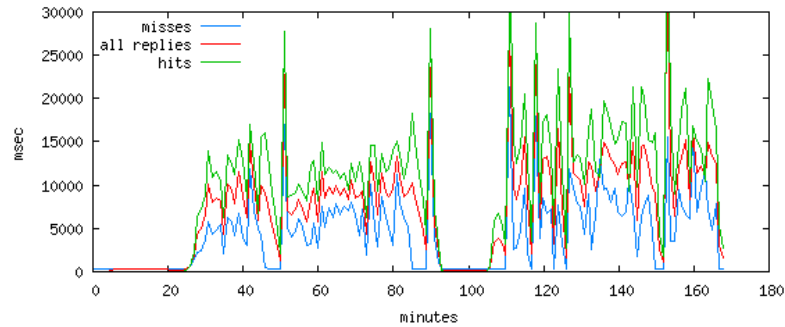


Figure 4.14: Response time trace, ATS

In Figure 4.15, the document and byte hit rates are shown. Out of 85.60% offered cachable data, the document hits rate for ATS was 54.4%, and the byte hit rates was 54.4% out of the 85.8% offered. The corresponding data for Varnish is a 47.5% document hit rate out of 85.8% offered and a byte hit rate of 47.7% out of 85.6% offered. The ATS hits rates also oscillate greatly throughout the experiment while the Varnish line is almost flat.

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

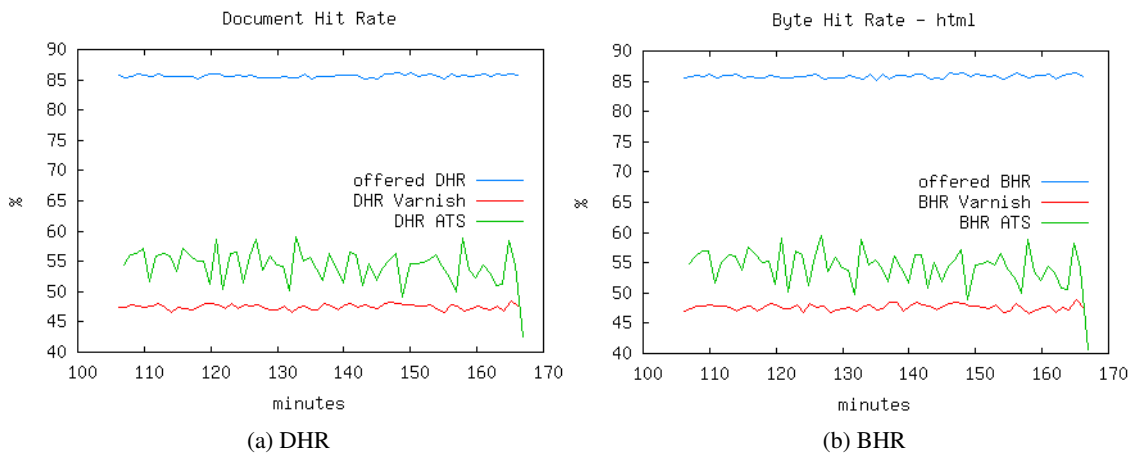


Figure 4.15: HTML content type document and byte hit rates

Figures 4.16 and 4.17 show the response time vs rate scatter plots for the HTML content type. The Varnish response rate is closely clustered around 400 request/s/sec with a nearly constant rate of 160-170 msec. On the other hand, the ATS response rate varied between 200 and 450 xact/sec, with response times from 2000 to 15000 msec and a few outliers in the range 20000-30000 (not visible in the figure).

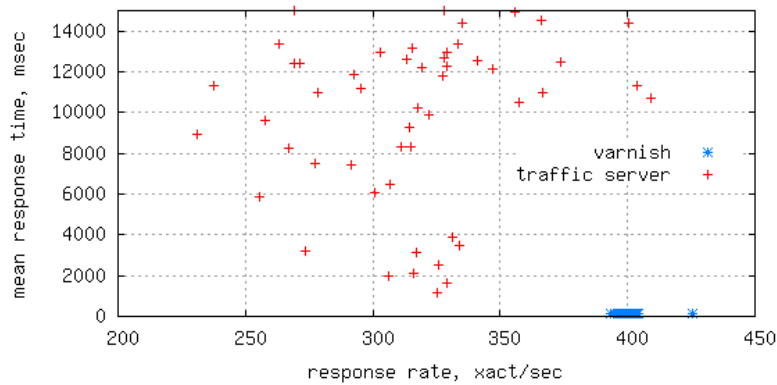


Figure 4.16: HTML content type response time vs response rate, client side

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

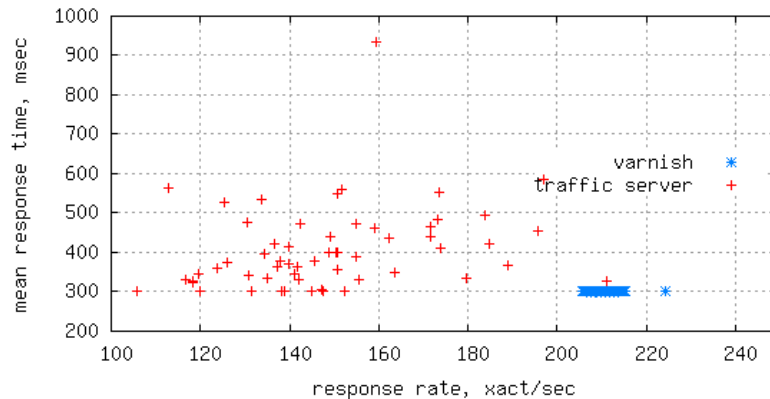


Figure 4.17: HTML content type response time vs response rate, server side

4.3.3 Download content type results

Table 4.3 shows the workload specifications used for HTML object type benchmarking. The phases are as described previously in Table 3.5.

The Download workload		
	Varnish	Apache Traffic Server
Working set size (MB)	50	50
Cache Size (GB)	1	1
Frozen WSS (GB)	1.71	3.19
Mean object size (KB)	294.9	294.9
Robot population	1000	1000
Content Type	Download	Download
Peak Rate	400 req/sec	400 req/sec

Table 4.3: Download workload defined and generated parameters

The workload described in table 4.3 contains only Download type content, which has relatively large files and is more challenging for cache servers to handle. The different frozen WSS values for the two servers come from the fact that the ATS fill phase last longer than for Varnish, allowing the working set size to grow to a larger final value.

Figure 4.18 shows that neither of the servers can attain the maximum request rate of 400 xact/sec. Nevertheless, ATS has better throughput (green line) than Varnish (blue line), with a rate of 155.7 xact/sec against 148.2. The bandwidth is higher for ATS as well, reaching an amount of 348.6 MB/sec out of the 353.04 offered rate against 336.5 MB/sec out of 343.95 offered for Varnish.

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

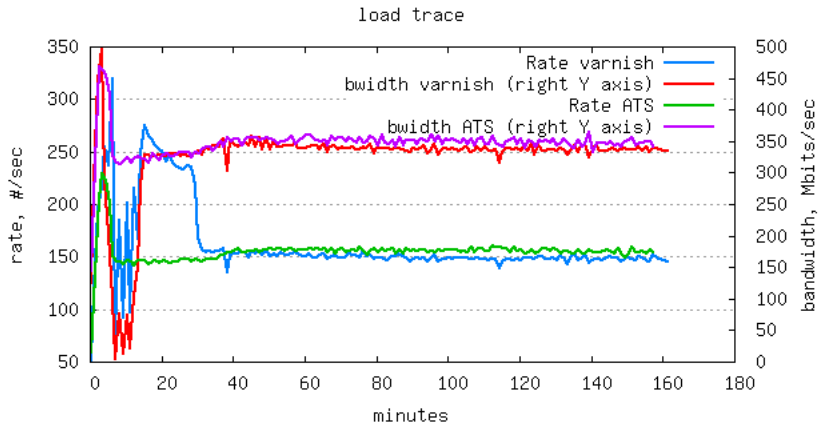


Figure 4.18: Throughput results for the Download content type

Figure 4.19 shows the response times for cache hits and misses. It shows that, unlike for the other tests, Varnish had clearly better performance. The blue and light blue lines show the Varnish hit and miss response times, which are just under 14000 msec. Interestingly, ATS, despite having longer response times than Varnish for this case, behaves the same as for earlier workloads, with a mean hit response time of 21000 msec and of 24000 msec for misses.

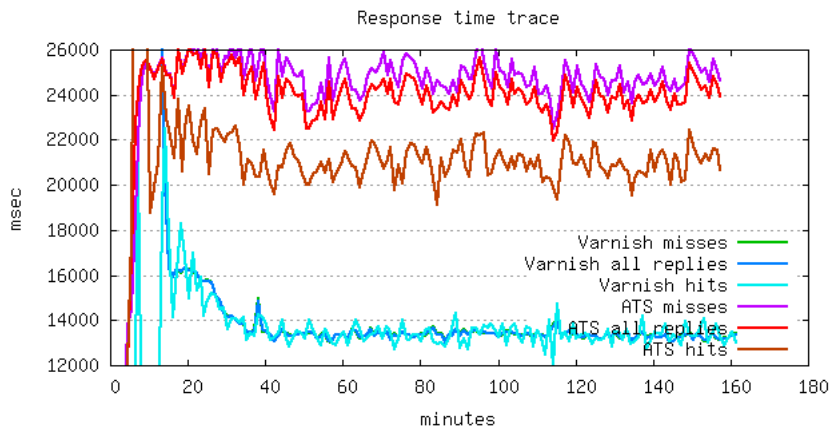


Figure 4.19: Response times for Download content type

Figure 4.20 shows that Traffic Server caches more than 20% of the load it receives while Varnish caches less than 10%. The same pattern is observed in the byte hit rate data.

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

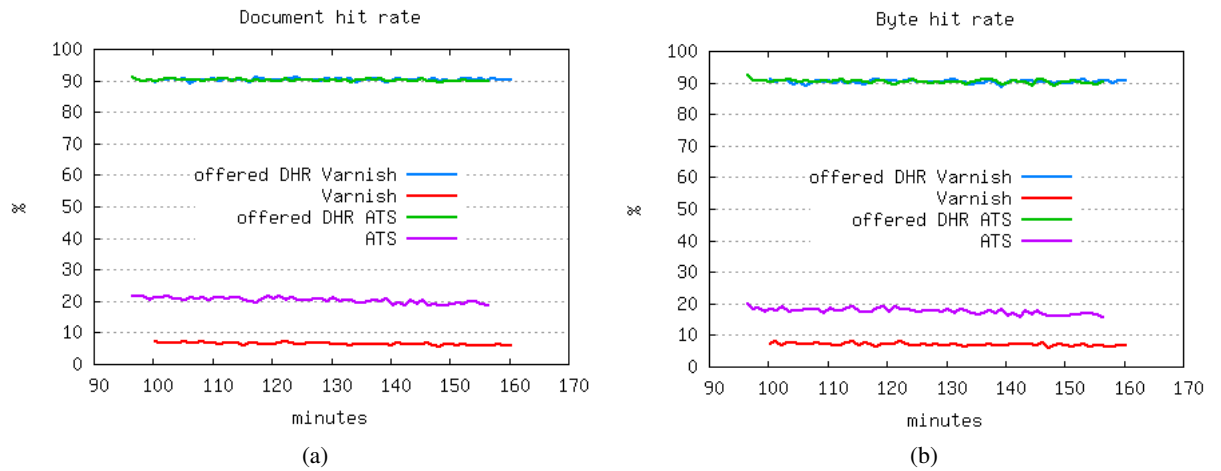


Figure 4.20: Download content type document and byte hit rates

Figure 4.21 illustrates how fast the proxy servers responded to requests during these experiments. The client side plot indicates that the response rate for Varnish varied from 140 to 154 xact/sec, with a mean response time of about 1300 msec. The results for ATS have a mean response time of 24000 msec, along with response rates in the range of 150-160 xact/sec.

On the server side, the response rate for ATS is less than for Varnish, but the response time is much higher than for Varnish. The ATS response rate varies from 120 to 128 xact/sec, with the response time landing between 8000 to 10000 msec. The response time for Varnish, however, is almost 10000 msec, with the response rate ranging between 130 to 140 xact/sec.

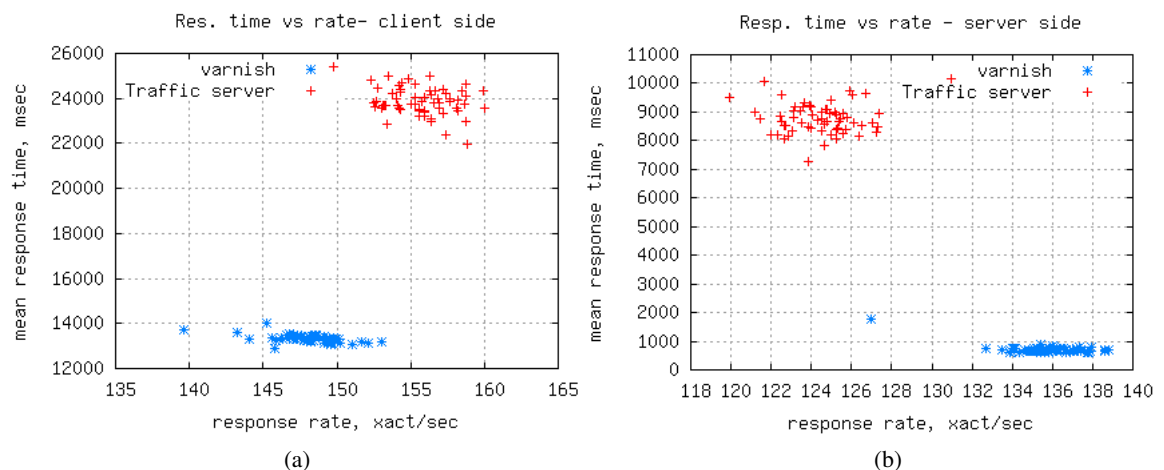


Figure 4.21: Download content type response time vs response rate

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

4.3.4 Summary and Discussion

In this section a short summary of the results attained from the baseline tests is presented and the important points are discussed. Figure 4.22 shows the results taken from the Image-only, HTML-only and Download-only content type experiments.

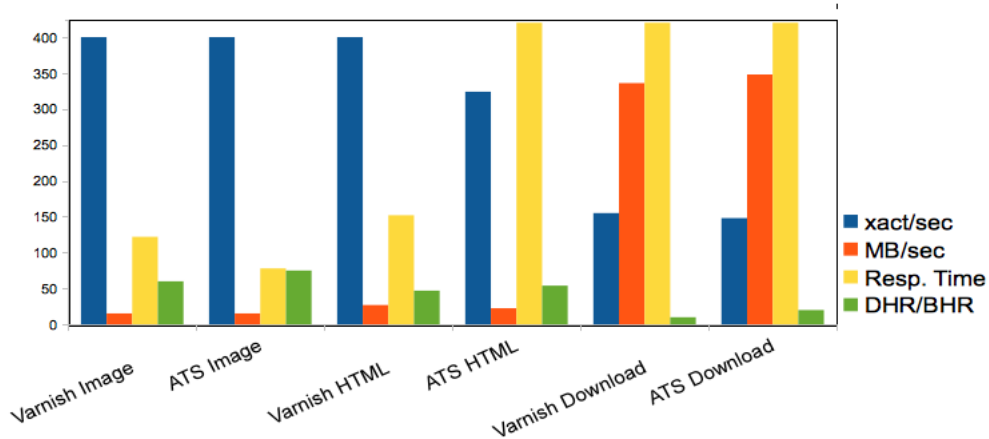


Figure 4.22: Summary of the Baseline tests

The results gained from each proxy in individual test are already taken in consider in their corresponding sections. In this section the discussion is most focused on comparing the results from different tests together. That means comparing/contrasting results from one individual proxy for all content types and then compared with results taken from the other proxy.

The blue lines illustrate that the number of transactions that proxy server can handle. For Varnish the number for Image and HTML types are almost equal with 400 xact/sec while the value is lower for Download content types as it is only 155 xact/sec.

ATS handles more Image content type than HTML with 400 vs 324 xact/sec for each. Like Varnish, The number for Download type is much lower for ATS as it is only 148 xact/sec. The bottom line is Varnish can response to the Image and HTML types almost the same while ATS is responding to more Image-only types than HTML-only types. Both of them handle lower rate of Download-only content type with nearly the same rate; 155 vs 148 xact/sec for Varnish and ATS respectively.

As the transaction/sec rate was the same for Varnish Image and HTML, the bandwidth utilization is also nearly the same with 29 vs 27 MB/sec(orange lines). The value is much more for Download type scenario as it is 336MB/sec. This huge difference comes from the difference in mean objects size for Download type which is objects with mean value 300KB while the size is 4 and 9KB

4.3. SINGLE CONTENT TYPE WORKLOAD RESULTS

for Image and HTML types. ATS shows more or less the same behaviour. The bandwidth usage is 15.3 vs 22.5 MB/sec for Image and HTML which is better for HTML. That can be correlated to the larger object size for HTML. The throughput for Download is 348 MB/sec which like Varnish case is much higher than Image and HTML. Varnish has higher bandwidth usage for Image and HTML cases which was predictable as Varnish has higher xact/sec rate for those scenarios. The interesting point in this case is that despite having lower transaction rate for ATS in Download case, it has a better bandwidth usage than Varnish; 348 vs 336 MB/sec. This mean that the objects that ATS has served have larger sizes.

The results for response times(yellow lines) are showing clear differences between performances. Varnish mean response time for Image and HTML are 122 and 162 msec. But the response time is considerably longer for Downloads with 13-14 seconds(the actual size is not shown in the figure because of large value and scaling problem). Despite having the same rate(xact/sec), it has longer response time for HTML which is explainable considering the mean size of objects. ATS response time is 78 msec for Images. Unlike Images , the response time is largely higher for HTML and Download cases which are 14 and 24 seconds respectively. The bottom line is that in Image scenario ATS has a better response time for Image-only content type while having worse results in HTML which is considerably higher(early signs of getting saturated) and double longer time in Download case. Varnish server could handle HTML content types as good as Image ones, but the performance reduced for Download test.

Document and Byte hit rates almost all of the time following very close pattern, for that reason they are categorized as one category. Varnish managed to cache 60% of 76% offered cachable data in Image test, while the number reduces to 47 out of 85 in HTML and 10 out of 90 % in Download cases. ATS follows the reducing pattern. It perfectly caches 75 out of 76% of offered cachable objects, but caches 54 and 20 % out of 85 and 90% of cachable items in HTML and Download scenarios respectively.

To summarize it all together and see which proxy performed better in individual cases the table 4.4 is made and shows the better performer in each case.

	Throughput	Response time	DHR/BHR	Resp. time/rate, client	Resp. time/rate, server
Best effort	ATS (slightly)	ATS (slightly)	ATS(slightly)	ATS	Varnish
Image	almost same	ATS	ATS	ATS	Varnish
HTML	Varnish	Varnish	ATS	Varnish	Varnish
Download	almost same	Varnish	ATS	Varnish	Varnish

Table 4.4: The better performer of each scenario

To summarize it all together, as we see in the table 4.4 in the Best effort scenario ATS performs slightly better. In other cases mostly Varnish has got a better

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

response time specially a much better hit response time. ATS in all cases out performs Varnish with better Document and Byte hit rates. In HTML case Varnish has a much better performance, and thr early signs of getting saturated is seen from ATS.

4.4 Results for the mixed content, increasing rate experiments

This section reports the results for the mixed content workload. Table 4.5 shows the phases included in this experiment. The key phases are highlighted. Their results will be discussed in separate subsections below.

Phase	Request Rate (% Peak Rate)	Recurrence (%)	Goal
warm	10 to 100	5	5 min
fill	100	5	2*Cache Size
link	fill to 100	95	5 min
top1	100	95	60 min
idle	10	95	0 min
inc	10 to 100	95	100 min
top2	100	95	10 min
cool	100 to 0	95	1 min

Table 4.5: The workload phases for the mixed content type experiment

The generated workload has two main parts which are highlighted in the table 4.5: The phase top1, which lasts for one hour and the phase inc, which lasts in 100 minutes. During the latter, the transaction rate will increase to 1600 requests/sec in order to find the saturation point and the bottlenecks of the proxies. The phase top1 use a constant rate of 320 requests/sec (20% of the peak rate), and both proxies can handle that rate without getting saturated. The workload parameters are defined in Table 4.6.

Mixed content workload phases		
	Varnish	Apache Traffic Server
Working set size (MB)	100	100
Cache Size (GB)	1.1	1.1
Frozen WSS (GB)	473	640
Mean object size (bytes)	4612.7 Bytes	4612.7 Bytes
Robot population	1600	1600
Content Type	all	all
Peak Rate (req/sec)	1600	1600

Table 4.6: Mixed content workload parameters

The distribution of content types for this workload is illustrated in Figure 4.23. The Other type refers to any other types of contents that can be found on the Internet other than the baseline types (Image, HTML, Download) which were discussed in the previous section.

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

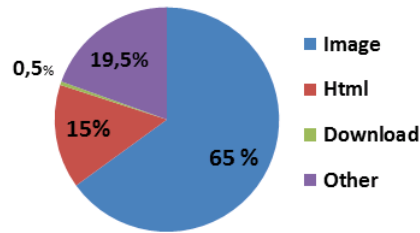


Figure 4.23: Mixed content workload content type distribution

4.4.1 Results for the top1 phase

In this phase of the test, the request rate is a constant rate of 320 xact/sec which is 20% of the peak rate of this experiment as a whole. The workload is a realistic workload, but delivered at a modest traffic rate which does not stress the proxy servers.

Figure reffig;topload demonstrates that the offered rates are the same (blue and green lines) for both proxy servers.

The bandwidth utilization for ATS is higher, with the mean throughput of 21.5 vs 19 MB/sec for Varnish. Comparing the results with the baseline results for throughput; here ATS performs slightly better, at the same level as the best effort workload results. In the baselines, the throughput is almost the same for the Image and Download contents, while in the Best Effort case ATS yields a bit better results.

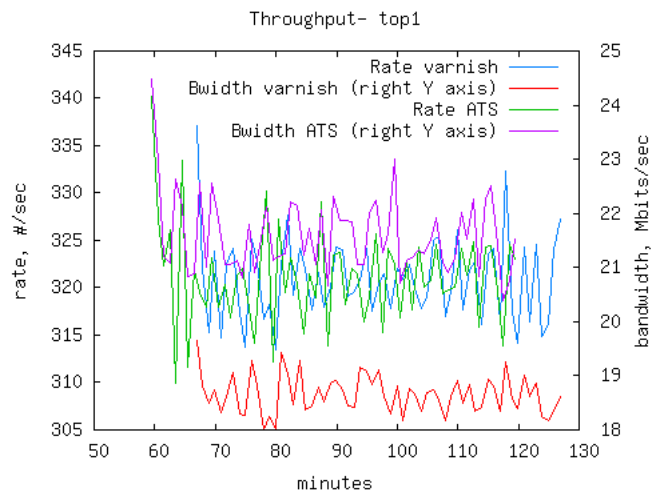


Figure 4.24: Modest request rate throughput

Figure 4.25 shows that the response time for misses is essentially the same (the blue and purple lines) at 300 msec. This number comes from the configured

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

"server think time" variable of Polygraph which is configured as 0.3 sec. The mean response time for all replies is slightly better for Varnish at 326 msec vs 332 msec for ATS. The reason for that is the better hit response time for Varnish, which is 1-2 msec against 15-17 msec for ATS. Recall the results from the baseline tests where ATS has a better response time for Images while Varnish has a better response time for the HTML and Download content types.

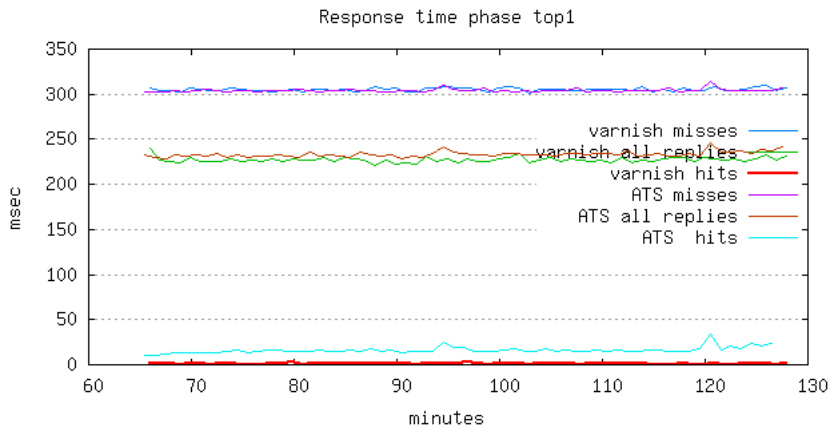


Figure 4.25: Response time results for top1 phase

Figure 4.26 shows the document and byte hit rates. Varnish's document hit rate is 27% of 74% offered and its byte hit rate is 27% of 75% offered cachable data; the corresponding values for ATS are 30.7% of 76.7% and 29.5% of 78%. The performance is again better for ATS, but the difference is not nearly as much. Comparing that with the baseline results reveals that ATS in all cases outperforms Varnish with respect to hit rate results. It can be speculated that the better hit rates for ATS are a result of utilizing multiple replacement policies in RAM replacement as well as the policy which is used for disk hits. The various algorithms used for evicting objects from the cache are shown in Table 2.3. As mentioned, ATS uses both recency-based and frequency-based algorithms as well as greedy size policies, including CLOCK policy implementation. Varnish on the other hand implements only a recency based policy (LRU).

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

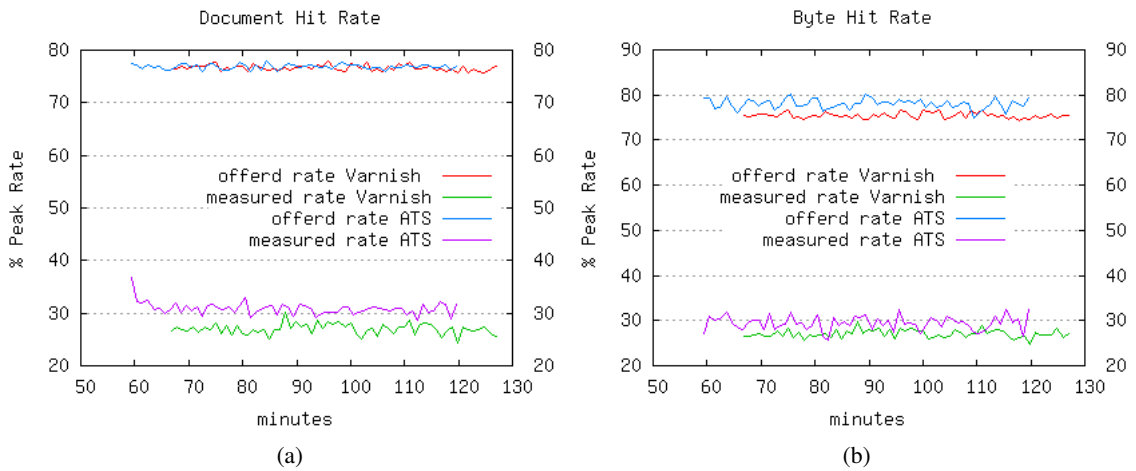


Figure 4.26: Document and byte hit rates, top1 phase

Figure 4.27a shows the distribution of response rates by response time. As previously explained, the X axis shows the response rate as seen from the client side. The Y axis shows the corresponding mean response time for that specific response rate. ATS values are plotted in red, and they are mostly concentrated between 315 and 325 milliseconds, with a mean response rate of 230-235 msec. For Varnish (in blue), the distribution of response rates are spread mostly between 315 and 325 xact/sec, and these results are very similar to ATS. The response time rate is a bit better for Varnish, with times between 220 and 230 msec against ATS's 230-235 msec.

Seen from the server side, the situation is a bit different. In Figure 4.27b, the response time and response rate are quite similar for both servers. They don't follow any particular pattern, and the response rate is mostly spread between 230 and 250 xact/sec for Varnish, and the same values apply for ATS. The response time range lies in a very narrow interval between 300 and 303 msec for both servers. Thus, they both have very similar distributions a slightly better response time for Varnish.

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

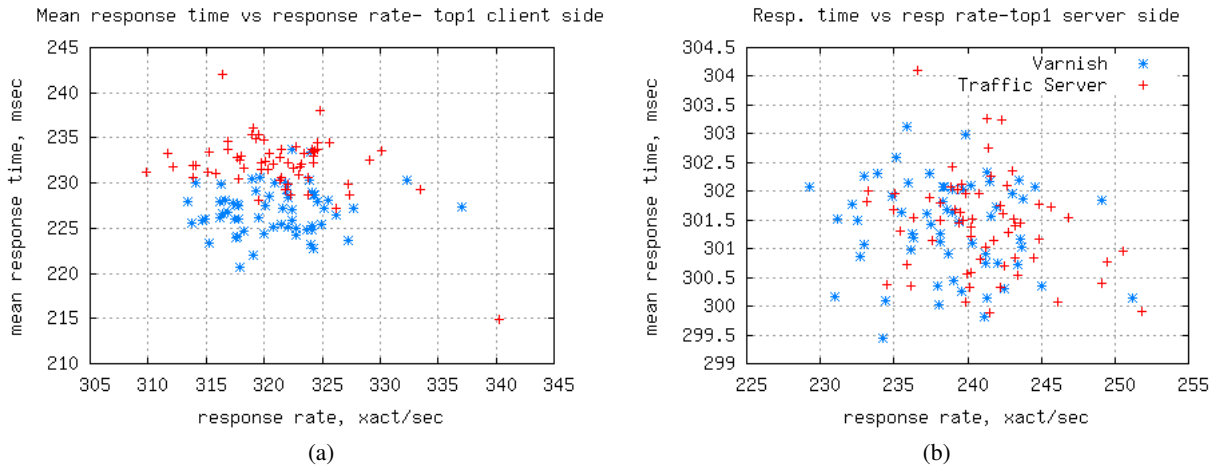


Figure 4.27: Response time vs response rate, *top1* phase

The conclusion for this phase is that both servers handle the load decently with only minor differences. ATS had a larger bandwidth throughput (21.5 vs 17 MB). Varnish had a better response time, specifically the hit rates' response time was considerably better (2 vs 15 msec), but the all replies response time was very close with a small difference in the favour of Varnish (228 vs 232 msec)

4.4.2 Increasing traffic rate phase inc

The inc phase is the second part of the mixed content test. It consists of an increasing the current rate, beginning at the current rate of 320 xact/sec (20% of the peak rate) to a maximum of 1600 xact/sec. The rate is gradually increased over 100 minutes so that the saturation point is found for servers under investigation. The workload parameters and content distribution are the same as for the top1 phase (see Table 4.6 and Figure 4.23).

After completing the top1 phase, the test enters the inc phase which immediately increases the rate of the requests.

Figure 4.28 shows the total timeline of the test. The phase inc (minute 125) starts just after phase top1 (minutes 65-125) ends. It is clearly seen that Varnish (the green and red lines) handle requests up to maximum rate (1600 xact/sec which is the peak rate). However, ATS shows signs of getting saturated at the rate 500 xact/sec (the blue line), and it cannot handle a rate higher than 800 xact/sec. Aside from a few spikes, it remains at a rate with the mean between 600 and 800 requests per second. The bandwidth does not exceed 60 MB/sec while Varnish can handle 1600 xact/sec and 100 MB/sec at its maximum performance.

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

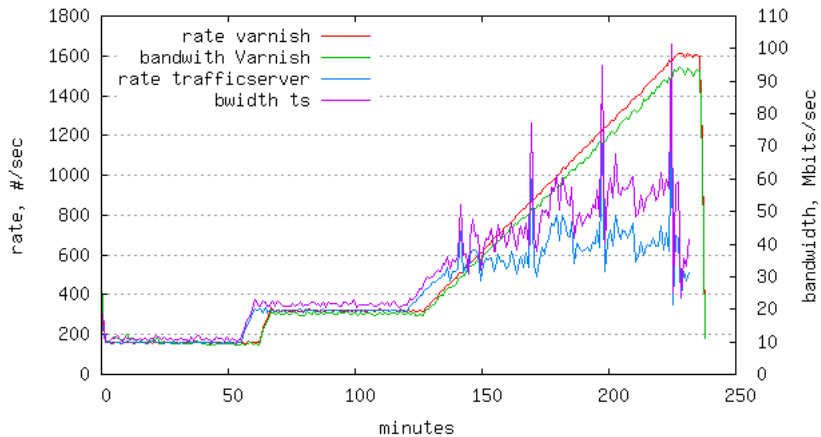


Figure 4.28: Mixed content workload phase inc

Figures 4.29 and 4.30 show the response time for Varnish and ATS respectively. Due to their huge differences, the results are presented in separate figures. The hits again have a very small values ranging from 1 msec to 7 msec at the end of the phase. A random spike is seen in all three types of response times, which means Varnish is showing the very early indications of getting to the saturation point. The miss response times which start at 300 msec grow during the experiment, reaching 316 msec. The same behavior applies to all replies responses, with a start point of 230 msec and a final value of 245 msec. This is the behavior for Varnish that was observed in all baseline tests except for the Download content type.

The response time for ATS in this phase is relatively high. The overall response time has a mean value of 8-10 seconds while the miss response time fluctuates between 6 and 10 seconds. The interesting point here again is the hit response time. Like HTML baseline test, where ATS got to saturation point, the hit response time in this phase is higher than the misses. This indicates that ATS cache handling is worsening under stress.

The results attained in this phase are similar to results obtained in HTML test, where ATS get saturated before Varnish and yields somewhat odd results with poor hit response time. Varnish performs as in the Image and HTML single content scenarios.

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

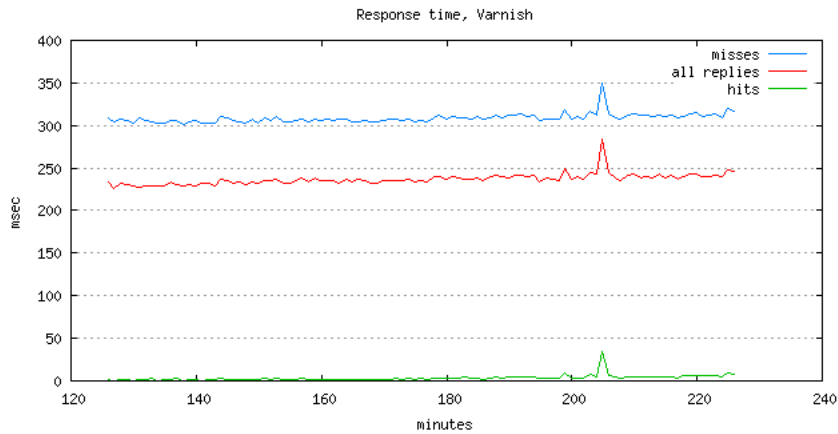


Figure 4.29: Varnish response time, inc phase

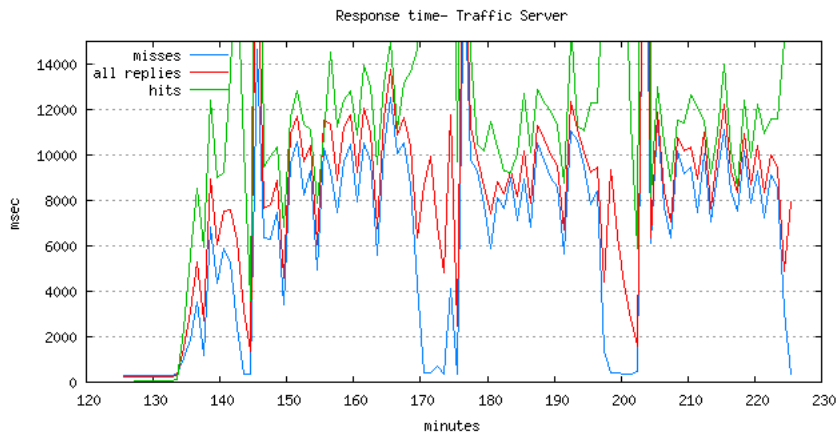


Figure 4.30: ATS response time, inc phase

Hit rates have quite different behaviour in this phase (see Figure 4.31). ATS performs better in the beginning, with 30% of 76% offered cacheable items, which is similar to all the baseline experiments, where ATS caches more data. The noticeable issue in this case is that, unlike the HTML-only case where ATS get saturated but still maintains higher hit rates than Varnish, here the hit rate reduces as the time goes forward. ATS ultimately caches only 20% of data, which is even worse than Varnish. Varnish during whole period maintains a constant rate of 25% caching of 76% offered cacheable data, exhibiting a stable hit rate value.

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

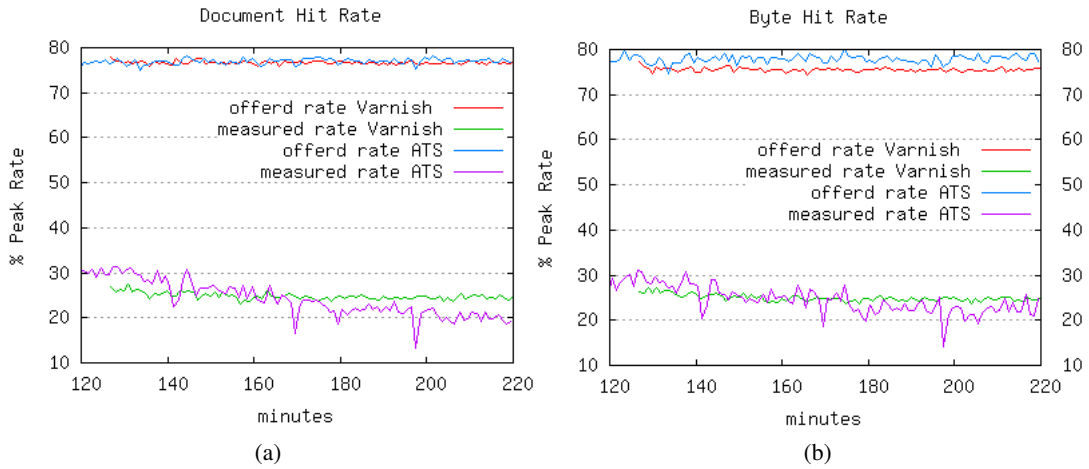


Figure 4.31: Hit rates during the *inc* phase

Figure 4.32 shows the response rate vs response time plot for this phase. Varnish's mean response time range is quite narrow, just as in the baselines, while ATS has the same response pattern as Varnish while the request rate is in the range below about 450 xact/sec. As the rate grows, there is no change in behavior seen from Varnish as it just continues with a linear response time. But ATS acts differently. As the rate increases above 450 xact/sec, the response time shows an initial exponential growth pattern, and afterwards it remains in the range between 800 and 1200 xact/sec on the client side. This is quite different from the way ATS reaches to the saturation point in the HTML scenario, where the response time is spread randomly from 2000 msec to 14000 msec.

Over the course of the experiment, Varnish's response rate range is wide because, unlike in the baseline experiments, the request continues to increase throughout this phase. The interesting issue for Varnish is that, despite large growth of the response rate, the response time remains almost unchanged.

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

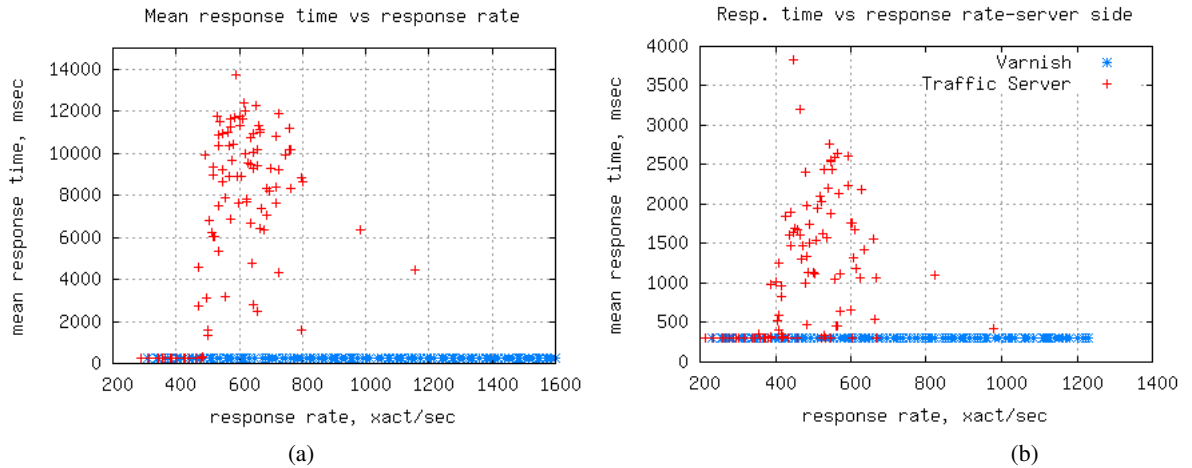


Figure 4.32: Response time vs response rate, *inc* phase

4.4.3 Summary and Discussion

Results in this experiment are divided in two distinct parts, corresponding to the top1 and inc phases. In the top1 phase, which simulates a typical scenario for a cache/web server, the results reveal that either of servers can handle the load in a reasonable manner, and they can respond to the full offered rate. The results are summarized in figure 4.33

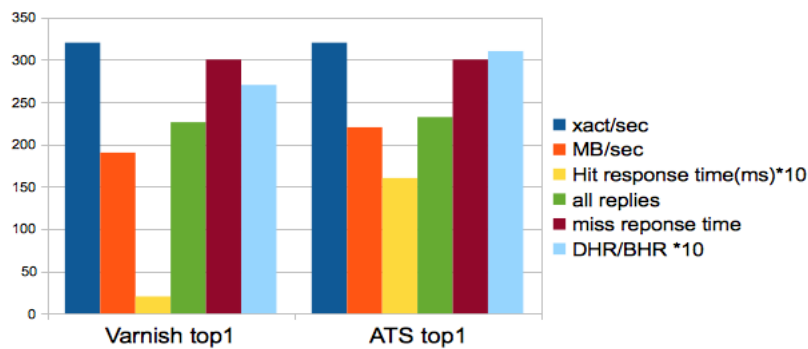


Figure 4.33: Performance during the constant rate phase

As the chart illustrates, the request/sec throughput (dark blue lines) for both servers is similar, and they both respond to full offered rate of 320 xact/sec. The bandwidth throughput for ATS is slightly better at 22 vs 19 MB (orange

4.4. RESULTS FOR THE MIXED CONTENT, INCREASING RATE EXPERIMENTS

lines).

The hit response time is considerably better for Varnish, with an average 3 msec against ATS's 16 msec. The average response times for misses and all replies are more or less similar.

When it comes to document and bte hit rates, ATS outperforms Varnish with better hit rates for either type.

Figure 4.34 shows the results attained from inc phase. Varnish yields better performance in all cases. The dark blue lines show the much higher xact/sec throughput for Varnish. Note that the graph shows the mean throughput over the whole period of the test: 100 minutes. In the final minutes of the phase, Varnish managed to handle a request rate of 1600 xact/sec (see Figure 4.28), which is much higher than the mean in Figure 4.34. ATS, on the other hand, can not handle rates higher than 600 xact/sec, beginning at minute 150 of the test, and remains at this level until the end of the test.

As a consequence of the higher request rate, the bandwidth throughput is larger for Varnish as well (the orange line). Despite huge differences in mean xact/sec, the bandwidth throughput exhibits smaller differences (cf. difference between the dark blue and orange lines).

The response time is the factor that shows the real difference from the client point of view. It can be observed in the chart that all three kinds of response times (hit, miss and all replies) have large values for ATS compared to those with Varnish. The peak response times for ATS are not shown in the figure due to having large values which exceed the scale. Overall, the mean response time for ATS is between 6-12 seconds while the corresponding numbers for Varnish are dramatically shorter, at only 300 msec at maximum.

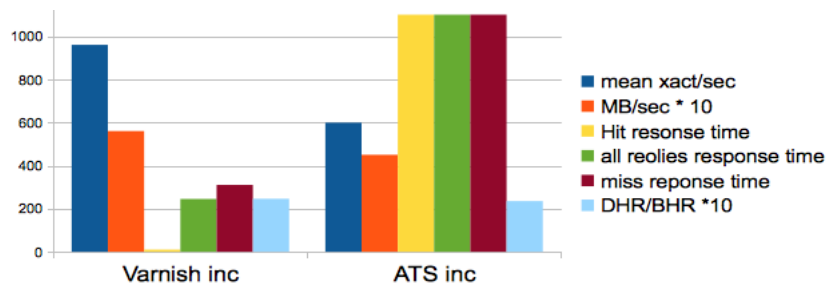


Figure 4.34: Performance during the increasing rate phase

Table 4.7 shows the results of each scenario from the above summarized results.

4.5. SURROGATE DATA RESULTS

	Throughput	Response time	DHR/BHR	Resp. time/rate, client	Resp. time/rate, server
top1	ATS (slightly)	Varnish(slightly)	ATS(slightly)	Varnish(slightly)	almost the same
linc	Varnish	Varnish	Varnish(slightly)	Varnish	Varnish

Table 4.7: Summary of results from the inc phase

4.5 Surrogate data results

In this section, the results of the surrogate data are discussed in the context of the preceding Polygraph results.

Best Effort Workload

Figure 4.35 shows the system CPU and memory usage during the Best Effort test. ATS uses both less CPU and a smaller percentage of memory throughout the test. Especially in terms of memory resources, ATS resource requirements remain relatively constant over the period of the test.

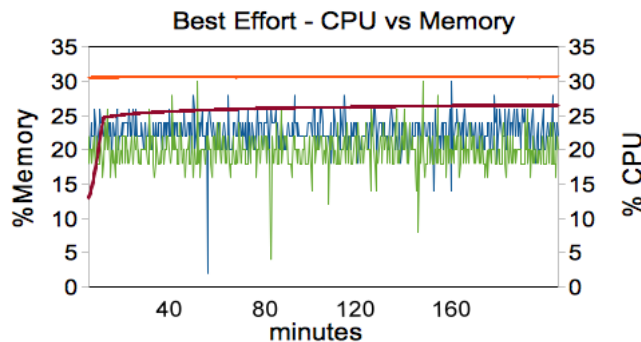


Figure 4.35: CPU and Memory Usage, Best Effort Workload

Varnish uses more CPU and more memory than ATS, and its CPU usage range is narrower than for ATS.

Figure 4.42 illustrates the Seekwatcher plots from the blktrace data recorded during the same experiment. Varnish sustains a higher I/O throughput rate from disk than ATS. The dip in the throughput graph occurs at the same time as the spike in the seek count plot. This is a logical result. These features occur during the experiment's idle phase. Varnish may have entered a garbage collection mode when the request rate dropped to 0, accounting for these artifacts. Otherwise, the Varnish seek rate is quite low, indicating that it is either serving data from RAM or sending requests to the back end server.

4.5. SURROGATE DATA RESULTS

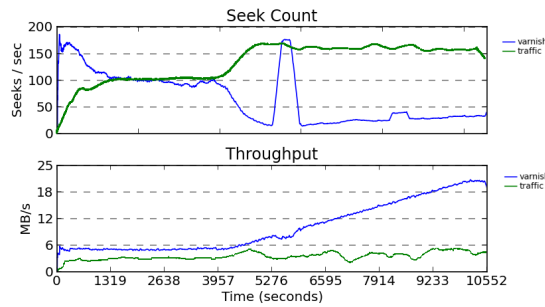


Figure 4.36: Disk I/O - mixed content type, increasing rate

Single Content Workloads

Figure 4.37 illustrates the CPU and main memory use for the three single content type workloads. ATS is relatively consistent in its memory usage; after a startup period, it uses between 25% and 34% of main memory across the various tests, with the maximum for HTML, the scenario for which it becomes saturated. Even after this occurs, ATS does not continually increase its memory use. In general, ATS is quite consistent in its memory usage across all of the experiments.

Varnish has completely flat memory usage for the HTML and Image test, although it uses more memory than ATS. For the Download content type, Varnish again uses more memory than ATS, and its usage oscillate very slightly during the course of the test.

For the Image and Download workloads, the CPU usage pattern for both servers is similar to that for the Best Effort workload, although the mean values shift as shown in Table 4.8. Note that the CPU percentage values are for 2 CPUs and so can exceed 100%. The idle phase for the Image experiment, during which server activity drops to zero, is excluded from the calculation of the mean.

	Best Effort	Image	Download
ATS	20	7	102
Varnish	23	9	99

Table 4.8: Mean %CPU usage for various workloads (2 cpus)

The CPU usage for the HTML workload again indicates the stress experienced by the ATS server, evidenced by the numerous usage spikes (green line). In contrast, Varnish's CPU usage remains steady at a quite low value of about 10% (again excluding the idle period).

4.5. SURROGATE DATA RESULTS

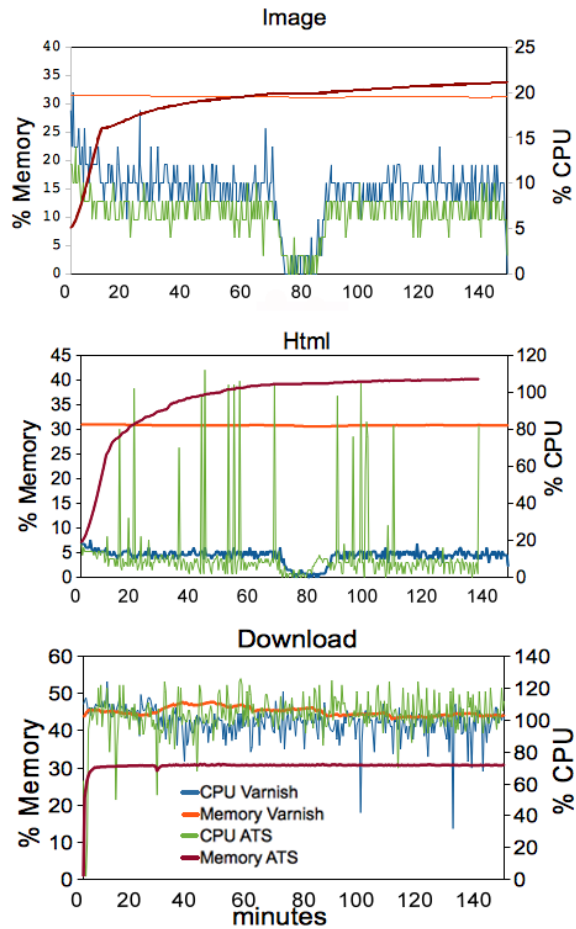


Figure 4.37: CPU/memory usage, single content type workloads

Figures 4.38 and 4.39 provide the Seekwatcher plots of the blktrace data for the HTML and Image content types. In both cases, there is again an upsurge in seeks on the part of Varnish during the idle phase; in contrast, ATS seeks decrease almost to zero over the same period. In addition, Varnish exhibits another large spike later in both runs. This behavior is hard to explain, but it recurs during experiment repetitions and is something that happens periodically with Varnish. It is certainly worth additional investigation.

4.5. SURROGATE DATA RESULTS

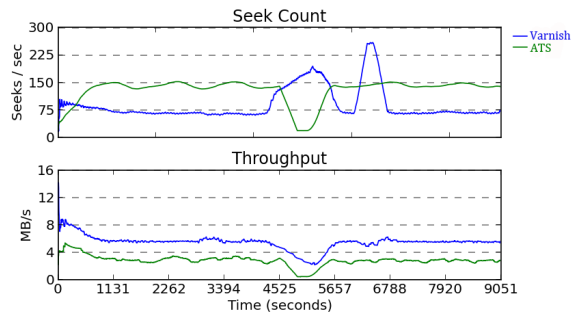


Figure 4.38: Disk I/O - HTML content type

The two servers' disk I/O throughput for these experiments show a very similar pattern to one another, although ATS again has a lower throughput level for both content types.

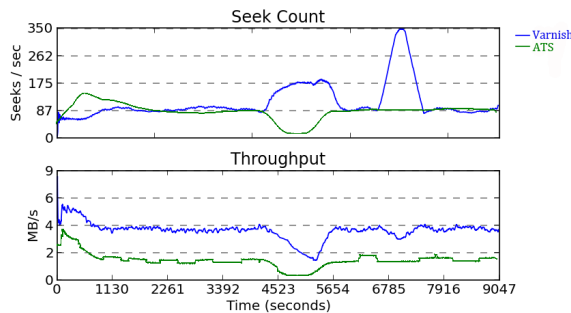


Figure 4.39: Disk I/O for Image content type

The Seekwatcher results for the Download content type are much more straightforward. The seek rates and I/O throughput are both flat, and the two servers have essentially the same behavior profiles. This is in line with their similar results for throughput and close results for other metrics from the Polygraph tests.

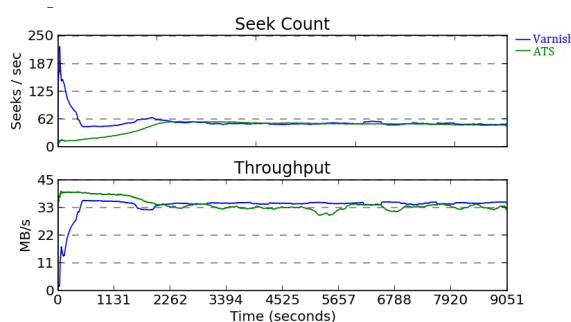


Figure 4.40: Disk I/O - Download content type

Mixed content results

Figure 4.41 shows the CPU and memory usage over the period of the mixed content workload experiment. The period prior to 70 minutes corresponds for the most part to the top1 phase. The two servers have essentially the same CPU usage over this period, with ATS being slightly more volatile. The memory usage again has ATS using several fewer percent of the CPU. These results are completely in agreement with those from the best effort and non-HTML single content tests.

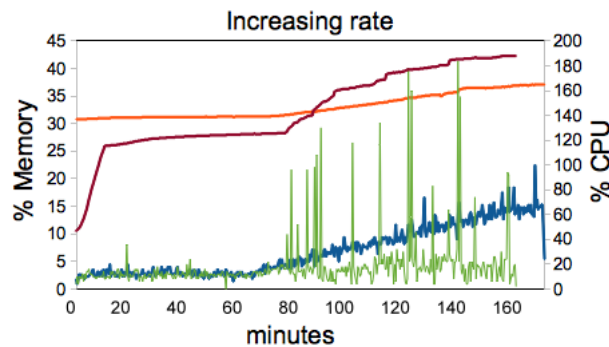


Figure 4.41: CPU/memory usage, mixed content type workload

After minute 70, the inc phase begins, and the data changes significantly. Varnish uses gradually increasingly more CPU and slightly more memory than during the top1 phase as the request rate rises.

ATS behavior is quite different. The CPU usage dramatically increases and oscillates wildly as the request rate rises. At times, the system is close to running out of CPU resource. Memory use also increases, rising to a level significantly over that of Varnish.

Figure 4.42 shows the Seekwatcher results for the same experiment. After the initial fill phase (from about the minute point), the two servers again have very similar results during the top1 phase. During the inc phase, Varnish throughput rises as it is capable of handling the increasing workload. In contrast, ATS's throughput remains at a lower level corresponding to the fraction of the offered workload that it is capable of handling.

4.5. SURROGATE DATA RESULTS

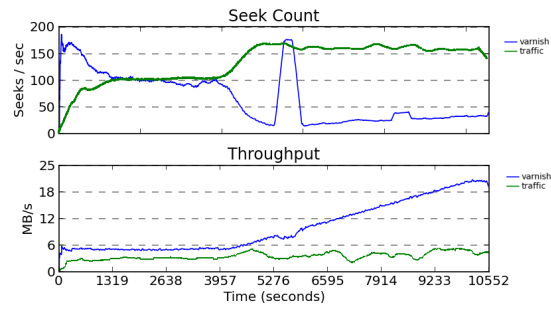


Figure 4.42: Disk I/O, mixed content type workload

The Varnish seek data again includes one very large and one smaller spike during the inc phase. However, this characteristic if unexplained behavior does not affect its I/O throughput results or its ability to handle the workload. Clearly, this data feature must correspond to some routine function of Varnish.

Chapter 5

Discussion

5.1 Summary of the results

The following points summarize the most important results of the experiments in this project.

- ATS performs slightly better in the Best Effort workload than Varnish although the difference is not very large.
- When ATS performs better than Varnish, it is on the more artificial, single content type workloads. For these workloads, the results can also be the same for the two servers.
- Varnish performs much better on the HTML content type workload where ATS becomes saturated.
- ATS has better document hits rates in almost all cases where it can handle the request rate. This means that it caches more of the offered cacheable data.
- Not considering the inc scenario where ATS is in the saturation state, the throughput for these products are mostly the same or slightly better for ATS.
- Varnish performs much better under the increasing request rate stress portion of the most realistic, mixed content workload.
- The most important metrics for reverse proxy servers are throughput and response time, and in most cases, Varnish's results are superior to those of ATS.
- In the results for response time versus response rate on the client side, performance is almost the same, if slightly better for Varnish. In the response time vs response rate on the server side, Varnish always performs better

5.2. ANALYSIS OF THE ATS SATURATION POINT

- When ATS is under stress, it tends to consume a great deal of CPU resources. This is only a serious problem if other work is intended to be done on the same server.
- Resource usage patterns are generally similar to the Polygraph performance results.

The most surprising point of the results is the much lower saturation point for ATS than Varnish: 500-600 xact/sec for ATS, while Varnish manage to response request rates of up to 1600 xact/sec.

5.2 Analysis of the ATS saturation point

Having a closer look at network metrics of each benchmark reveals that the number of pending connections when the request rate increases in the inc phase grows exponentially with a long-tailed distribution for ATS (see Figure 5.1). In early minutes of the inc phase, ATS gets a high number of concurrent HTTP connections. Thirty minutes after starting the phase, the concurrent HTTP connections reach 500,000 connections, and this tendency continues until the end of the test, ultimately reaching 2.5 million concurrent connections.

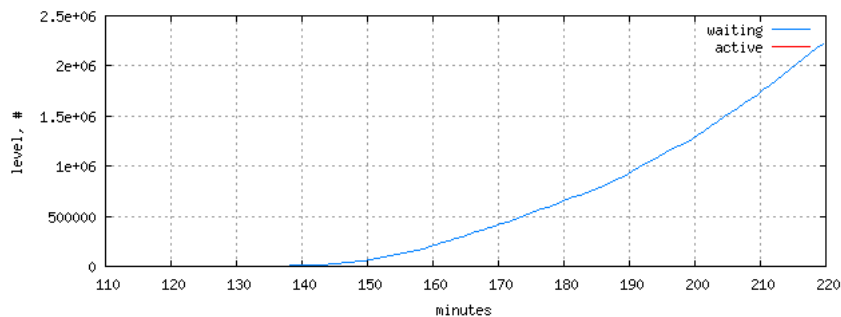


Figure 5.1: Concurrent HTTP connections for ATS

At the same time, considering the total number of TCP/HTTP connections (see Figure 5.2), ATS cannot handle more than 6000 concurrent connections. ATS remains at constant level of 6000 while the request rate continue to increase. This is the reason for exponential growth of waiting connections for ATS.

5.2. ANALYSIS OF THE ATS SATURATION POINT

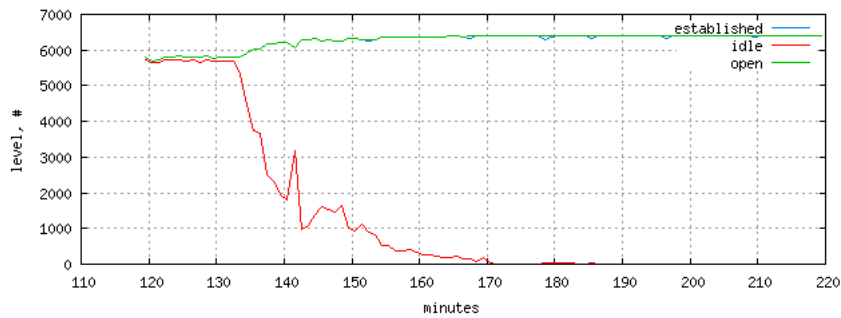


Figure 5.2: Concurrent HTTP/TCP connections for ATS

Figures 5.3 and 5.4 also show the network metrics for Varnish, which exhibits numbers that are dramatically lower than for ATS. At the start of phase inc in minute 125, the number of concurrent HTTP connections increases linearly, proportional to the request rate increase. The connections do not reach to a level more than 500 concurrent connection even at the highest request rates toward the end of period. The pending connections do not exceed 150 connection at any given time.

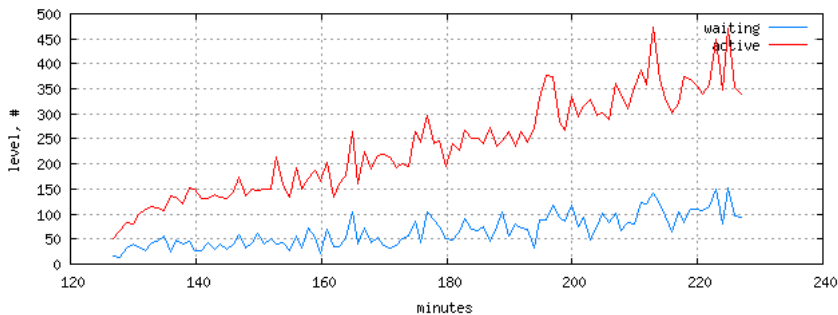


Figure 5.3: Concurrent HTTP connections for Varnish

Figure 5.4 shows that the number of TCP connections grow linear and proportional to request rate as well, starting with 1000 concurrent TCP connections and reaching a peak of 3500 at the end of the experiment.

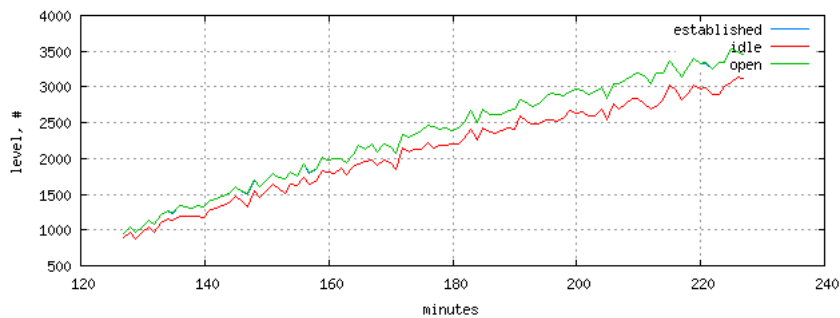


Figure 5.4: Concurrent HTTP/TCP connections for Varnish

5.3 Pitfalls and Issues with the Server Software

The process of installing of Varnish is quite straightforward. The only issue here is that Debian and other distribution repositories may not have the latest version, and earlier versions are significantly more unstable. For example, the first version used for this research had a memory leakage bug causing Varnish to crash. In order to mitigate the problem, it is better to install the product from the `varnish-cache.org` repository or install from the source.

There are some potential pitfalls in configuring Varnish as well.

- By default, Varnish aggressively caches the objects and ignore the client header "Cache-Control: no-cache" which it shouldn't. This resulted in too many Polygraph "hit on uncachable request" errors. This problem was solved by adding the flag `-t` (timeout) when running `varnishadm` daemon and assigning the value of zero, which tells Varnish to finish the operation – in this case, caching the object – immediately.
- The same problem occurs with the Reload HTTP type request which Varnish serves from the cache instead of via a fresh object from the backend server. The attempt to solve this was unsuccessful. Modifications to `vcl_fetch()` didn't solve the problem but instead caused another problem which led to memory leakage. The errors resulting from this bug were ignored in the analysis.
- Another problem with configuring Varnish is that the amount of the memory that it is using for indexing pages and other "housekeeping" activities is not controllable, meaning that Varnish utilizes as much memory as it wants. By experience it was observed that Varnish regularly used up to 30% of the whole memory capacity and ATS was not using more than 8-10%. In order to make the two test configurations as identical as possible, an additional 1 GB of RAM was given to ATS, which resulted actual memory use that was much closer to Varnish. The surrogate tests revealed

5.4. DIFFICULTIES WITH THE POLYGRAPH TOOL

that ATS used more memory than was expected during the mixed content workload, by about 10%.

Apache Traffic Server installation process was also straightforward. However the initial configuration of ATS was more tricky than Varnish. The ATS package is configured as a forward proxy server. In order to make it a reverse proxy server, further configuration was needed (discussed in Section 3.6).

When using a reverse proxy server in front of an actual web server, the proxy acts as if it is the actual web server. Thus the requests should be sent to the proxy rather than (actual backend) web server. It means that the proxy is transparent to clients, and clients send their requests thinking they are sending traffic to the real web server.

When trying to benchmark ATS using Polygraph, the requests didn't get through, and Polygraph generated an error for each single request and the test failed. By further investigation and sniffing the request packets with tcpdump[66], it was determined that Polygraph was generating requests destined for the actual web server which did not match ATS's remap rules. By changing the remap rules to what is presented in section 3.6, this problem was solved. The remap rule was changed to remap requests destined to "the web server" to again to the "web server"!

5.4 Difficulties with the Polygraph tool

Web Polygraph is an advanced tool which supports many features of the HTTP protocol, but there are quite a lot challenges using that.

First of all, the lack of good documentation. Although there is a website intended to be a resource for users, the website does not actually provide sufficient information. What it seems to be trivial for Polygraph developers is not for other users. The existing documentations do not give proper knowledge to identify and understand the key concepts using the tool. Concepts are introduced with little explanation and without working examples. It was a time consuming process to try and fail with various changes in the configuration files to identify the impact of each change, as it takes relatively long time to fill the caches and run a test.

The second problem with Polygraph is that it does not tolerate even small time differences between systems that it works with. After detecting some few hundred millisecond time drift between them, it kills the client process and the test fails. The problem shows itself when trying to send higher rates of requests. This problem however could be partially solved by installing an NTP (Network Time Protocol) server on the proxy server machine and NTP clients on the client and web server machines.

5.5. OPEN ISSUES AND FUTURE WORK

However, despite having NTP installed, the problem persisted with high request rates. The other measurement which was taken to alleviate the problem was adding more robots with a lower request rate for each, so that the final peak request rate is equal to previous one, with the difference that short time drifts do not kill the process.

An other issue with Polygraph is that it does not provide the raw numeric data for its results, instead producing HTML pages with graphical output. To solve the problem, some code was added to the ReportFigure.cc file of the Polygraph source code in order to store a local copy of processes data.

5.5 Open Issues and Future Work

Benchmarking web caches contains relatively wide range of elements to be taken into account. There are quite many other things which can be done in conjunction with this project. One of them is using HTTP traffic with range headers and different types of HTTP requests. Since considerable time was spent learn how Polygraph generates loads, there was insufficient time to design and configure all of the desirable workload types.

SSL traffic is common on the Internet, but could not be tested during this research. SSL traffic can be generated by Polygraph, but Varnish does not support SSL traffic. The argument that Varnish developers make is that the whole Varnish code is smaller than, for instance, the OpenSSL code, making it unreasonable to introduce mechanisms for SSL support into Varnish. What is recommended to overcome this shortcoming is to use an SSL proxy like nginx or pound in front of Varnish. Apache Traffic Server does supports SSL traffic.

Another possibility for testing is generating SSL workloads, and traffic which needs to be authenticated. However, neither Varnish nor Apache Traffic Server support authentication of traffic. ATS removed this feature in 2008 when the product was converted to an open source product. For Varnish, the developers' philosophy is that authentication should be handled by the backend (private communication).

The scenarios for all these tests have been identical and fair but performed in an isolated network. Thus the results for response times are shorter than in reality. Because of the identical scenarios, the best results in this project would be the best in real networks and the worst results would be worst in real networks. However, to get results which are closer to real numbers, one would need to inject artificial latencies in the network. One can use software like Dummysnet for that purpose.

Another thing that might be helpful is to collect additional system data in order to investigate more the unexpected disk I/O usage gained from the surrogate tests. Metrics like buffer cache sizes and block I/O in/out operations to/from memory. Such research could begin using a common tool like vmstat. Use of the buffer cache by proxy servers is also an interesting area of future investigaiton.

Chapter 6

Conclusion

This thesis describes research into reverse proxy server performance. It has accomplished the following:

- After learning about its working and functionality, several experimental workloads were created using the Web Polygraph tool.
- These workloads were run using the Varnish and Apache Web Server proxy servers configured to be as similar as possible.
- Additional system metrics were collected simultaneously with the Polygraph runs.
- These results were extracted and analyzed in order to compare the behavior of the two servers.

Overall, ATS is shown to have better cache hit rates, thus reducing the traffic to the backend web server. While this an important goal for forward proxy servers, in reverse proxy servers, the important issue is to accelerate the data traffic rather than trying to save bandwidth to the backends. Varnish shows better response time performance, which is desirable. In addition, Varnish can tolerate much higher request rate pressures. Based on this research, Varnish seems the more promising reverse proxy server.

Bibliography

- [1] <http://cacheoff.ircache.net/>.
- [2] B.D. Davison. A web caching primer. *Internet Computing, IEEE*, 5(4):38–45, jul/aug 2001.
- [3] Cisco visual networking index: Forecast and methodology, 2010-2015.
- [4] *Web Caching*. O'Reilly & Associates Inc, 2001.
- [5] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. *lecture notes in computer science*, 2001.
- [6] <http://www.web-polygraph.org/>.
- [7] <http://www.hpl.hp.com/research/linux/httpperf/>.
- [8] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, sept. 1982.
- [9] Helen Thomas Anindya Datta, Kaushik Dutta and Debra VandeMeer. A comparative study of alternative tier caching solutions to support dynamic web content acceleration.
- [10] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, and K. Ramamritham. Accelerating dynamic web content generation. *Internet Computing, IEEE*, 6(5):27–36, sep/oct 2002.
- [11] Improving web server performance by caching dynamic data. In *USENIX symposium on Internet technologies and systems*.
- [12] H. Zhu and T. Yang. Class-based cache management for dynamic web content. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1215–1224 vol.3, 2001.
- [13] Geoff Huston and Telstra. Web caching. *The internet Protocol Journal*, 1999.
- [14] <http://www.mozilla.org/>.
- [15] <http://windows.microsoft.com/en-us/internet-explorer/products/ie/home>.

BIBLIOGRAPHY

- [16] <http://www.apple.com/safari/>.
- [17] B. Ciciani, F. Quaglia, P. Romano, and D. Dias. Analysis of design alternatives for reverse proxy cache providers. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems*, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on, pages 316 – 323, oct. 2003.
- [18] Content distribution network. <http://akamai.com>.
- [19] Jun Wu and K. Ravindran. Optimization algorithms for proxy server placement in content distribution networks. In *Integrated Network Management-Workshops, 2009. IM '09. IFIP/IEEE International Symposium on*, pages 193 –198, june 2009.
- [20] Bo Li, M.J. Golin, G.F. Italiano, Xin Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1282 –1290 vol.3, mar 1999.
- [21] Lili Qiu, V.N. Padmanabhan, and G.M. Voelker. On the placement of web server replicas. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1587 –1596 vol.3, 2001.
- [22] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. RFC 2186 (Informational), September 1997.
- [23] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.
- [24] C. Aggarwal, J.L. Wolf, and P.S. Yu. Caching on the world wide web. *Knowledge and Data Engineering, IEEE Transactions on*, 11(1):94 –107, jan/feb 1999.
- [25] B. Krishnamurthy and C.E. Wills. Proxy cache coherency and replacement-towards a more complete picture. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 332 –339, 1999.
- [26] Stefan Podlipnig and Laszlo Böszörményi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, dec 2003.
- [27] S Jin and A Bestavros. Greedydual web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174 – 183, 2001.
- [28] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A Fox. Caching proxies: Limitations and potentials. Technical report, Virginia Polytechnic Institute and State University, 1995.
- [29] Mimi Pitkow, James Edward ; Recker. A simple yet robust caching algorithm based on dynamic access patterns. Technical report, Georgia Institute of Technology, 1994.

BIBLIOGRAPHY

- [30] Mike Reddy & Graham P. Fletcher. Intelligent web caching using document life histories: A comparison with existing cache management techniques. Technical report, J228, School of Computing, University of Glamorgan, Pontypridd, 1998.
- [31] J. Zhang, R. Izmailov, D. Reininger, and M. Ott. Web caching framework: analytical models and beyond. In *Internet Applications*, 1999. IEEE Workshop on, pages 132–141, aug 1999.
- [32] A. Vakali. Lru-based algorithms for web cache replacement. In Kurt Bauknecht, Sanjay Madria, and Gajntner Pernul, editors, *Electronic Commerce and Web Technologies*, volume 1875 of *Lecture Notes in Computer Science*, pages 409–418. Springer Berlin / Heidelberg.
- [33] Saied Hosseini-Khayat. Investigation of generalized caching. PhD thesis, St. Louis, MO, USA, 1998. UMI Order No. GAX98-07761.
- [34] Analyzing performance of partitioned caches for the WWW, 1998.
- [35] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. *Performance Evaluation*, 39(1):149–164, 2000.
- [36] Variable QoS From Shared Web Caches: User-Centered Design and Value-Sensitive Replacement., 2001.
- [37] Noritaka Osawa, Toshitsugu Yuba, and Katsuya Hakozaiki. Generational replacement schemes for a www caching proxy server. In Bob Hertzberger and Peter Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 940–949. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0031665.
- [38] MCGREGOR T. CHANG, C.-Y. and G HOLMES. The lru* www proxy cache document replacement algorithm. In .
- [39] Improving effectiveness of Web caching. In *Recent Advances in Distributed Systems*, chapter ., 2000.
- [40] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. *SIGCOMM Comput. Commun. Rev.*, 26(4):293–305, August 1996.
- [41] Igor Tatarinov. An efficient lfu-like policy for web caches. Technical report, Computer Science Department, North Dakota State University, Wahpeton, ND., 1998.
- [42] Kai Cheng and Y. Kambayashi. Lru-sp: a size-adjusted and popularity-aware lru replacement algorithm for web caching. In *Computer Software and Applications Conference*, 2000. COMPSAC 2000. The 24th Annual International, pages 48–53, 2000.

BIBLIOGRAPHY

- [43] Pei Cao Pei Cao. Cost-aware www proxy caching algorithms. Technical report, University of Wisconsin-Madison, University of California-Irvine, 1997.
- [44] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Evaluating server-assisted cache replacement in the web. In Gianfranco Bilardi, Giuseppe Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, Algorithms – ESA 98, volume 1461 of Lecture Notes in Computer Science, pages 1–1. Springer Berlin / Heidelberg, 1998. 10.1007/3-540-68530-8_26.
- [45] Qiang Yang, H.H. Zhang, and Hui Zhang. Taylor series prediction: a cache replacement policy based on second-order trend analysis. In System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on, page 7 pp., jan. 2001.
- [46] Nicolas Niclausse, Zhen Liu, and Philippe Nain. A new efficient caching policy for the world wide web, 1997.
- [47] Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8):977 – 986, 1997. <ce:title>Papers from the Sixth International World Wide Web Conference</ce:title>.
- [48] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Trans. Netw.*, 8(2):158–170, April 2000.
- [49] Hyokyung Bahn, Kern Koh, S.H. Noh, and S.M. Lyul. Efficient replacement of nonuniform objects in web caches. *Computer*, 35(6):65 –73, jun 2002.
- [50] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance Evaluation*, 46(2):125 – 137, 2001. <ce:title>Advanced Performance Modeling</ce:title>.
- [51] K. Psounis and B. Prabhakar. A randomized web-cache replacement scheme. In INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1407 –1415 vol.3, 2001.
- [52] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.
- [53] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005.
- [54] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.
- [55] Apache traffic server. <http://trafficserver.apache.org/>.

- [56] Apache review2009. <http://ostatic.com/blog/guest-post-yahoos-cloud-team-open-sources-traffic-server>.
- [57] Apache traffic server wiki. <https://cwiki.apache.org/confluence/display/TS/RamCache/>.
- [58] Varnish reverse proxy. <https://www.varnish-cache.org/>.
- [59] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 1, pages 126 –134 vol.1, mar 1999.
- [60] Steven Glassman. A caching relay for the world wide web. Computer Networks and ISDN Systems, 27(2):165 – 173, 1994. <ce:title>Selected Papers of the First World-Wide Web Conference</ce:title>.
- [61] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In Parallel and Distributed Information Systems, 1996., Fourth International Conference on, pages 92 –103, dec 1996.
- [62] Norifumi Nishikawa, Takafumi Hosokawa, Yasuhide Mori, Kenichi Yoshida, and Hiroshi Tsuji. Memory-based architecture for distributed www caching proxy. Computer Networks and ISDN Systems, 30(1&A7):205 – 214, 1998. <ce:title>Proceedings of the Seventh International World Wide Web Conference</ce:title>.
- [63] <http://linux.die.net/man/8/blktrace/>.
- [64] <http://oss.oracle.com/mason/seekwatcher/>.
- [65] <http://www.gnuplot.info/>.
- [66] <http://www.tcpdump.org/>.

6.1 Appendix: The automation and Surrogate scripts

```
----- The polystart.pl script -----
1  #!/usr/bin/perl
2  use Getopt::Std;
3  use strict "vars";
4  use Net::SCP;
5
6  my $VERBOSE = 0 ;
7  my $DEBUG = 0;
8
9  my $optString = 'vdhc:s:l:a:';
10 getopts($optString,\my %opt) or usage() and exit 1;
11
12 if ( $opt{'h'} ) {
13     usage();
14     exit 0;
15 }
16
```

6.1. APPENDIX: THE AUTOMATION AND SURROGATE SCRIPTS

```
17 $VERBOSE = 1 if $opt{'v'};
18 $DEBUG = 1 if $opt{'d'};
19
20 my $client = "192.168.0.3";
21 my $server = "192.168.0.2";
22
23 verbose("Verbose is enabled");
24 debug("Debug is enabled");
25 my @accelerators = (trafficserver);
26 my @logs = (be4);
27
28 my $log = $opt{'l'} ? $opt{'l'} : "test";
29 my $accelerator = $opt{'a'};
30
31 print "\nhave you checked the working set storage, unique world ..etc parameters?.."; sleep 10;
32 verbose("flush varnish cache...");
33
34 foreach $log (@logs){
35     verbose("flushing the caches");
36     '/etc/init.d/varnish start'; sleep 5;
37     'varnishadm -T 127.0.0.1:6082 ban.url .;sleep 5;
38     '/etc/init.d/varnish stop';sleep 5;
39     verbose("flush traffic server cache...");
40     '/etc/init.d/trafficserver stop'; # sleep 5; 'traffic_server -Cclear';sleep 5;
41
42     foreach $accelerator (@accelerators){
43
44         verbose("killing previous server and client processes, if any ...");
45         'ssh -l root $client "pkill poly"';
46         'ssh -l root $server "pkill poly"';
47
48         system("/etc/init.d/varnish stop; /etc/init.d/trafficserver stop");
49         my $log_server = "$accelerator"."_server"."$log"."log";
50         my $log_client = "$accelerator"."_client"."$log"."log";
51
52         my $s_config = "configs/$log"."pg";
53         my $c_config = "configs/$log"."pg";
54
55         verbose("log is $log");
56         verbose("config file is $c_config and $s_config");
57         verbose("Accelerator is: $accelerator");
58         verbose("server log file : $log_server");
59         verbose("client log file : $log_client");
60
61         verbose("Setting ulimits to 500000");
62         system("ulimit -n 500000");
63         'ssh -l root $server "ulimit -n 500000";
64         'ssh -l root $client "ulimit -n 500000";
65         print "\nhave you checked the working set storage, unique world ..etc parameters?.."; sleep 10;
66         verbose("Flushing the system caches...");
67         system("sync; echo 3 > /proc/sys/vm/drop_caches");
68         'ssh -l root $client "sync; echo 3 > /proc/sys/vm/drop_caches";
69         'ssh -l root $server "sync; echo 3 > /proc/sys/vm/drop_caches";
70
71
72         sleep 10;
73
74         verbose("Flushing network connections caches...");
75         system("/etc/init.d/networking restart"); sleep 50;
76         'ssh -l root $client "/etc/init.d/networking restart";
77         'ssh -l root $server "/etc/init.d/networking restart"; sleep 50;
78
79         my $run_server = "ssh -l root $server '(ulimit -n 65000 ; /usr/local/polygraph/bin/polygraph-server
80         --unique_world off --local_rng_seed 501 --config $s_config --verb_lvl 10 --dump errs --log results,$log_server)";
81
82         my $run_client = "ssh -l root $client '(ulimit -n 65000 ; /usr/local/bin/polygraph-client --unique_world off
```

6.1. APPENDIX: THE AUTOMATION AND SURROGATE SCRIPTS

```
83     --local_rng_seed 601 --config $c_config --verb_lvl 10 --dump errs --log results/$log_client");
84
85     verbose("starting $accelerator ...");
86     my $start_accelerator = "(ulimit -n 100000 ; /etc/init.d/$accelerator start)";
87
88     system("$start_accelerator");
89     sleep 5;
90
91     verbose("running server and client processes.... may take long time ...");
92     open(SR, "$run_server | ") or die "could not run server process: $\n";
93     verbose("poly server is started...");
94     sleep 5;
95
96     open(CL, "$run_client | ") or die "could not run server process: $\n";
97
98     my $directory = "$accelerator$log";
99
100    ## start vmstat and blktrace
101
102    my $blkt = "./blktrace.pl -s $accelerator -o $directory ";
103    my $vstat = "./vmstat.pl -f $directory";
104    my $top = "./top.pl $accelerator $log";
105    my $adad = 'ssh -l root $client ./awkp.pl';
106    verbose("adad is $adad..");
107
108    my $afill = 1;
109
110    sleep 10;
111    verbose("poly-graph is running ... if client finished, waiting til server is done with cold phase");
112    while( $adad > 2){
113        $adad = 'ssh -l root $client ./awkp.pl';
114        verbose("adad is $adad sleep 20...\n");
115        my $lr = "polygraph-lr results/$accelerator"._client_"$log".log | grep i-link | wc -l";
116
117        if($afill eq "1"){
118            my $inc = 'ssh -l root $client "$lr" '; verbose("inc is $inc");
119            if($inc > 3){
120                open(BLK, "$blkt | ") or die "could not run blktrace $\n";
121                verbose("blktrace started...");
122                open(VS, "$vstat | ") or die "could not run vmstat $\n";
123                verbose("vmstat started..."); sleep 10 ;
124                open(TP, "$top | ") or die "could not run top $\n";
125                verbose("top started...");
126                $afill = 0 ;
127            }
128        }
129        sleep 20;
130    }
131    verbose("poly-graph finished ... killing poly processes(if any)...");
132    'ssh -l root $client "pkill poly";
133    'ssh -l root $server "pkill poly";
134
135    verbose("kill vmstat, blktrace and stop $accelerator..");
136    'pkill vmstat' ;
137    'pkill blktrace ' ;
138
139    verbose("stopping $accelerator");
140    system("/etc/init.d/$accelerator stop");
141
142    sleep 10;
143    if( $accelerator eq "trafficserver") { $accelerator = "traffic_server"; }
144    verbose("copying the files ...");
145    my $csv_file = "$directory"."csv";
146    my $blktrace0 = "$directory"."blktrace.0";
147    my $blktrace1 = "$directory"."blktrace.1";
```

6.1. APPENDIX: THE AUTOMATION AND SURROGATE SCRIPTS

```
149     my $stop_file = "top.".$accelerator".$log".".csv";
150     my $png0 = "$blktrace0".".png";
151     my $png1 = "$blktrace1".".png";
152
153     verbose("running seekwatcher ...");
154     system("seekwatcher -t $blktrace0 -o $png0");
155     system("seekwatcher -t $blktrace1 -o $png1");
156
157     system("mkdir $directory; mv $csv_file $stop_file $directory");
158     system("mkdir $directory; mv $csv_file $blktrace0 $blktrace1 $stop_file $directory");
159
160     verbose("copying the files, preparing for reporter ...");
161
162     my $scp1 = Net::SCP->new( "$server", "root");
163     my $scp2 = Net::SCP->new( "$client", "root");
164
165     $scp1->get("results/$log_server") or die $scp1->{errstr};
166     $scp2->put("$log_server") or die $scp2->{errstr};
167     'ssh -l root $client "mv $log_server results/$log_server"';
168     system("rm $log_server");
169
170     $scp2->put("$png1") or die $scp2->{errstr};
171     $scp2->put("$png0") or die $scp2->{errstr};
172
173     system("mv $png1 $png0 $directory");
174
175
176     my $reporter_command = "/usr/local/bin/polygraph-reporter --report_dir /var/www/$directory
177     --label"." \"$directory\" " " . "results/$log_server results/$log_client" ;
178
179     verbose("reporter command : $reporter_command");
180     'ssh -l root $client "$reporter_command"';
181
182     'ssh -l root $client "mkdir /var/www/$directory/";'
183     'ssh -l root $client "mkdir /var/www/$directory/figures";'
184     'ssh -l root $client "mkdir /var/www/$directory/seekwatcher";'
185     'ssh -l root $client "cp -r /tmp/polyrep/$directory/figures/* /var/www/$directory/figures/";'
186     'ssh -l root $client "cp -r /tmp/polyrep/$directory/* /var/www/$directory/";'
187     'ssh -l root $client "mv $png1 $png0 /var/www/$directory/seekwatcher";'
188
189
190 }
191 verbose("wait until TIME_WAIT connections are terminated...");
192 sleep 60;
193 verbose("\n\n done with $accelerator.\n\nkilling previous server and client processes, if any ...");
194 'ssh -l root $client "pkill poly";'
195 'ssh -l root $server "pkill poly";'
196 sleep 60;
197 }
198
199 print "\n\n\n*****test finished *****\n\n";
200
201
202 sub top{
203     my $ts = $_[0];
204     my $log = $_[1];
205
206     my $pid = pid($ts); verbose("pid is $pid");
207
208 }
209 sub usage {
210     print "Usage: \n";
211     print "-h Usage \n";
212     print "-v Verbose \n";
213     print "-s server config file\n";
214     print "-c client config file \n";
```


6.1. APPENDIX: THE AUTOMATION AND SURROGATE SCRIPTS

```
215     print "-l log file name \n";
216     print "-a accelerator \n";
217     print "- \n";
218     print "./script [-d] [-v] [-h] \n";
219 }
220
221 sub verbose {
222     print $_[0]."\n" if $VERBOSE;
223 }
224
225 sub debug {
226     print $_[0]."\n" if $DEBUG;
227 }
```

The blktrace.pl script

```
1  #!/usr/bin/perl
2
3  use Getopt::Std;
4  use strict "vars";
5
6  my $VERBOSE = 0 ;
7  my $DEBUG = 0;
8
9  my $optString = 'vhs:o:';
10 getopts($optString,\my %opt) or usage() and exit 1;
11
12 my $ser = $opt{'s'} ;
13 my $output = $opt{'o'};
14 my $dev;
15 if ($ser eq "varnish"){ $dev = "/dev/sda1" }
16 if ($ser eq "trafficserver"){ $dev = "/dev/sda1" }
17 my $command = "blktrace -d $dev -b 4096 -n 20 -o $output";
18 system("$command");
19
20 if ( $opt{'h'} ) {
21     usage();
22     exit 0;
23 }
24
25 $VERBOSE = 1 if $opt{'v'};
26
27 verbose("Verbose is enabled");
28 sub usage {
29     print "Usage: \n";
30     print "-h Usage \n";
31     print "-v Verbose \n";
32     # print "-d Debug \n";
33     print "./script [-d] [-v] [-h] \n";
34 }
35
36 sub verbose {
37     print $_[0]."\n" if $VERBOSE;
38 }
39
40 sub debug {
41     print $_[0]."\n" if $DEBUG;
42 }
```

The top.pl script

```
1  #! /usr/bin/perl
2  use Getopt::Std;
3  use strict "vars";
4
5  my $VERBOSE = 0 ;
6  my $DEBUG = 0;
```

6.1. APPENDIX: THE AUTOMATION AND SURROGATE SCRIPTS

```
7
8 my $optString = 'vdh';
9 getopts($optString,\my %opt) or usage() and exit 1;
10
11 if ( $opt{'h'} ) {
12     usage();
13     exit 0;
14 }
15
16 $VERBOSE = 1 if $opt{'v'};
17 $DEBUG = 1 if $opt{'d'};
18
19 verbose("Verbose is enabled");
20 debug("Debug is enabled");
21
22 my $ts = $ARGV[0];
23 my $log = $ARGV[1];
24
25 if ($ts eq "trafficserver"){
26     $ts = "traffic_server";
27 }
28 if ($ts eq "varnish"){
29
30 }
31
32 my $OUT_FILE = "top.".$ts".$log".csv";
33 open(OUT, "> $OUT_FILE") or die "cant open $OUT_FILE $!\n";
34 print OUT "Time,%CPU,%MEMORY\n";
35
36 my $tmp = "top.".$ts";
37
38 my $tim;
39
40 while( `ssh -l root 192.168.0.3 "ps aux | grep polygraph" | wc | awk '{print $1}'` > 2){
41
42     my $pid = findpid();
43     system("top -b -n 1 -p $pid > $tmp ");
44     sleep 3; $tim = time;
45     open( F, "< $tmp ") or die "can not open $tmp ... $!\n";
46     while( my $line = <F>){
47
48         if( $line =~ /^.*$pid\s+nobody.*\D\s+(\d+)\s+(\d+\.\d+).*$ ){
49             print OUT "$tim,$1,$2\n";
50
51         }
52         if( $line =~ /^.*$pid\s+traffics.*\D\s+(\d+)\s+(\d+\.\d+).*$ ){
53
54             print OUT "$tim,$1,$2\n";
55
56         }
57     }
58     close F; sleep 20;
59 }
60
61 sub findpid {
62
63     my $command = "ps aux | grep $ts ";
64     open(TOP, "$command | ") or die "cant run top $!\n";
65
66     my $pid;
67     if($ts eq "varnish"){
68
69         while( my $line = <TOP>){
70
71             if( $line =~ /^.*nobody\s+(\d+).*$.$ ){
```

6.2. APPENDIX: THE BASELINE, CONFIGURATION FILES

```
73     $pid = $1; return $pid;
74   }
75 }
76 }
77
78 }elseif($ts eq "traffic_server"){
79
80   while( my $line = <TOP>){
81     if( $line =~ /^.*107\s+(\d+).*traffic_server.*$/ ){
82       $pid = $1; return $pid;
83     }
84   }
85 }
86 }
87 close TOP;
88 }
89
90 sub usage {
91   print "Usage: \n";
92   print "-h Usage \n";
93   print "-v Verbose \n";
94   print "-d Debug \n";
95   print "./script [-d] [-v] [-h] \n";
96 }
97
98 sub verbose {
99   print $_[0]."\n" if $VERBOSE;
100 }
101
102 sub debug {
103   print $_[0]."\n" if $DEBUG;
104 }
```

6.2 Appendix: The baseline, configuration files

```
----- The IMAGE.gp -----
1  #include "/usr/local/share/polygraph/workloads/include/contents.pg"
2  #include "/usr/local/share/polygraph/workloads/include/phases.pg"
3  rate PeakRate = 400/sec;
4
5  // Fill rate (must be between 10% and 100% of peak request ratePeakRate)
6  rate FillRate = 100% * PeakRate;
7
8  // the two standard working set sizes are 100MB and 1GB
9  size WSS = 100MB;
10
11 // Cache size affects the duration of the "fill" phase below
12 // Use the sum of RAM and cache disks physical capacity
13 size CacheSize = 1GB;
14
15 // Random body with a length from 50KB to 100KB.
16 Content cntSimple = {
17   size = unif(50KB, 100KB);
18 };
19
20 // describe WebAxe-1 server
21 Server S = {
22   kind = "WebAxe-1-srv";
23
24 //   contents = [ cntImage: 50%, cntHTML: 15%, cntDownload: 0.5%, contentims: 20%, cntOther ];
25 contents = [ cntImage];//
26 direct_access = [ cntImage];//
27
```

6.2. APPENDIX: THE BASELINE, CONFIGURATION FILES

```
28     xact_think = norm(0.3sec, 0.1sec);
29     pconn_use_lmt = zipf(16);
30     idle_pconn_tout = 15sec;
31     http_versions = [ "1.1" ]; //newer agents use HTTP/1.1 by default
32
33 };
34
35 // where the simulated servers and robots will be located
36 // these ips will need adjustments based on your local environment,
37 // working set size, and request rate!
38 addr[] srv_ips = ['192.168.0.2:9000' ];
39 addr[] rbt_ips = ['192.168.0.3' ** 1000 ];
40 addr[] proxy_ip = ['192.168.0.1:80' ];
41
42
43
44 Robot R = {
45     kind = "WebAxe-1-rbt";
46     origins = srv_ips;
47     http_proxies = proxy_ip;
48
49     recurrence = 95%;
50     embed_recur = 100%;
51     pop_model = { pop_distr = popUnif(); };
52
53     req_rate = 0.4/sec;
54     pconn_use_lmt = zipf(64);
55     open_conn_lmt = 4; // open connections limit
56
57     //launch_win = 2.5min; // avoid burst of requests at start
58     http_versions = [ "1.1" ]; // newer agents use HTTP/1.1 by default
59
60
61     // 10% of requests have a Range header
62     req_types = ["Basic"]; // : 70%, "lms200" :10%, "lms304" : 10%, "Reload" ]; //, "Range": 20% ];
63
64 };
65
66 // compute actual request rate
67 PeakRate = 400/sec;
68 //PeakRate = count(rbt_ips)*R.req_rate;
69
70
71 /* phases */
72
73 Phase phWarm = {
74     name = "warm";
75     goal.duration = 5min;
76     load_factor_beg = 0.1;
77     load_factor_end = FillRate/PeakRate;
78     log_stats = false;
79 };
80
81 Phase phFill = {
82     name = "fill";
83     goal.fill_size = 2*CacheSize;
84     recur_factor_beg = 5%/95%;
85 };
86
87 Phase phLink = {
88     name = "link";
89     goal.duration = 10min;
90     load_factor_end = 1.0;
91     recur_factor_end = 1.0;
92 };
93
```

6.2. APPENDIX: THE BASELINE, CONFIGURATION FILES

```
94 Phase phTop1 = { name = "top1"; goal.duration = 60min; };
95 Phase phDec = { name = "dec"; goal.duration = 5min; load_factor_end = 0.1; };
96 Phase phIdle = { name = "idle"; goal.duration = 10min; };
97 Phase phInc = { name = "inc"; goal.duration = 5min; load_factor_end = 1.0; };
98 Phase phTop2 = { name = "top2"; goal.duration = 60min; };
99
100 // build schedule using some well-known phases and phases defined above
101 schedule(phWarm , phFill, phLink,phTop1, phDec, phIdle, phInc, phTop2, phCool);
102 //schedule(phWarm);
103
104 working_set_length(WSS/(10KB*FillRate)/5%/80%);
105
106 S.addresses = srv_ips;
107 R.addresses = rbt_ips;
108 use(S, R);
```

The mixed increasing rate configuration file pipee.gp

```
1
2 #include "/usr/local/share/polygraph/workloads/include/contents.pg"
3 #include "/usr/local/share/polygraph/workloads/include/phases.pg"
4
5
6 // Request rate is determined by the number of robots and is computed
7 // later; setting request rate here has no effect
8 rate PeakRate = 1600/sec;
9
10 // Fill rate (must be between 10% and 100% of peak request rate PeakRate)
11 rate FillRate = 10% * PeakRate;
12
13 // the two standard working set sizes are 100MB and 1GB
14 size WSS = 100MB;
15
16 // Cache size affects the duration of the "fill" phase below
17 // Use the sum of RAM and cache disks physical capacity
18 size CacheSize = 1100MB;
19
20 // describe WebAxe-1 server
21
22 Content contentims = {
23   client_behavior.req_types = ["lms200": 50%,"lms304"];
24   client_behavior.req_methods = ["POST"];
25   size = exp(11KB);
26   cachable = 80%;
27 };
28
29
30 Server S = {
31   kind = "WebAxe-1-srv";
32
33   contents = [ cntImage: 65%, cntHTML: 15%, contentims:10% , cntDownload: 0.5%,cntOther ];
34   direct_access = [ cntHTML,cntDownload, cntOther ];
35
36   xact_think = norm(0.3sec, 0.1sec);
37   pconn_use_lmt = zipf(16);
38   idle_pconn_tout = 15sec;
39   http_versions = [ "1.1" ]; // newer agents use HTTP/1.1 by default
40 };
41
42 // where the simulated servers and robots will be located
43 // these ips will need adjustments based on your local environment,
44 // working set size, and request rate!
45 addr[] srv_ips = [ '192.168.0.2:9000' ];
46 addr[] rbt_ips = [ '192.168.0.3' ** 1600 ];
47 addr[] proxy_ip = [ '192.168.0.1:80' ];
48
49 // describe WebAxe-1 robot
```

6.2. APPENDIX: THE BASELINE, CONFIGURATION FILES

```
50 Robot R = {
51   kind = "WebAxe-1-rbt";
52   origins = srv_ips;
53   // http_proxies = proxy_ip;
54
55   recurrence = 95%;
56   embed_recur = 100%;
57   interests = [ "public": 90%, "private" ];
58   pop_model = { pop_distr = popUnif(); };
59
60   req_rate = 1/sec;
61
62   req_types = ["Basic": 70%, "lms200": 10%, "lms304": 10%, "Reload"];
63   req_methods = ["POST", "GET": 70%, "HEAD"];
64   post_contents = [contentims];
65
66   pipeline_depth = zipf(13%);
67   pconn_use_lmt = zipf(64);
68   open_conn_lmt = 4; // open connections limit
69
70   //launch_win = 2.5min; // avoid burst of requests at start
71   http_versions = [ "1.1" ]; // newer agents use HTTP/1.1 by default
72 };
73
74 // compute actual request rate
75 PeakRate = count(rbt_ips)*R.req_rate;
76
77
78 /* phases */
79
80 Phase phWarm = {
81   name = "warm";
82   goal.duration = 5min;
83   load_factor_beg = 0.1;
84   load_factor_end = FillRate/PeakRate;
85   log_stats = false;
86 };
87
88 Phase phFill = {
89   name = "fill";
90   goal.fill_size = 2*CacheSize;
91   recur_factor_beg = 5%/95%;
92 };
93
94 Phase phLink = {
95   name = "link";
96   goal.duration = 5min;
97   load_factor_end = 0.2;
98   recur_factor_end = 1.0;
99 };
100
101 Phase phTop1 = { name = "top1"; goal.duration = 60min; };
102 Phase phDec = { name = "dec"; goal.duration = 5min; load_factor_end = 0.1; };
103 Phase phIdle = { name = "idle"; goal.duration = 10min; };
104 Phase phInc = { name = "inc"; goal.duration = 100min; load_factor_end = 1.0; };
105 Phase phTop2 = { name = "top2"; goal.duration = 10min; };
106
107 // build schedule using some well-known phases and phases defined above
108 schedule(phWarm, phFill, phLink, phTop1, phInc, phTop2, phCool);
109
110 // convert WSS in terms of "volume" to WSS in terms of units of "time"
111 working_set_length(WSS/(10KB*FillRate)/5%/80%);
112
113 // assign agents (servers and robots) to their hosts
114 S.addresses = srv_ips;
115 R.addresses = rbt_ips;
```

6.2. APPENDIX: THE BASELINE, CONFIGURATION FILES

116
117
118

```
// commit to using these servers and robots
use(S, R);
```

The content file- content.gp

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

```
ObjLifeCycle olcStatic = {
    length = const(2year); // two year cycle
    variance = 0%; // no variance
    with_lmt = 100%; // all responses have LMT
    expires = [nmt + const(0sec)]; // everything expires when modified
};

// object life cycle for "HTML" content
ObjLifeCycle olcHTML = {
    length = logn(7day, 1day); // heavy tail, weekly updates
    variance = 33%;
    with_lmt = 100%; // all responses have LMT
    expires = [nmt + const(0sec)]; // everything expires when modified
};

// object life cycle for "Image" content
ObjLifeCycle olcImage = {
    length = logn(30day, 7day); // heavy tail, monthly updates
    variance = 50%;
    with_lmt = 100%; // all responses have LMT
    expires = [nmt + const(0sec)]; // everything expires when modified
};

// object life cycle for "Download" content
ObjLifeCycle olcDownload = {
    length = logn(0.5year, 30day); // almost no updates
    variance = 33%;
    with_lmt = 100%; // all responses have LMT
    expires = [nmt + const(0sec)]; // everything expires when modified
};

// object life cycle for "Other" content
ObjLifeCycle olcOther = {
    length = unif(1day, 1year);
    variance = 50%;
    with_lmt = 100%; // all responses have LMT
    expires = [nmt + const(0sec)]; // everything expires when modified
};

// PolyMix-1 content
Content cntPolyMix_1 = {
    kind = "polymix-1"; // just a label
    mime = { type = undef(); extensions = []; };
    size = exp(13KB);
    obj_life_cycle = olcStatic;
    cachable = 80%;
};

Content cntImage = {
    kind = "image";
    mime = { type = undef(); extensions = [ ".gif", ".jpeg", ".png" ]; };
    obj_life_cycle = olcImage;
    size = exp(9KB);
    cachable = 80%;
    checksum = 1%;
};

Content cntHTML = {
    kind = "HTML";
```

6.2. APPENDIX: THE BASELINE, CONFIGURATION FILES

```
62     mime = { type = undef(); extensions = [ ".html" : 60%, ".htm" ] };
63     obj_life_cycle = olcHTML;
64     size = exp(8.5KB);
65     cachable = 90%;
66     checksum = 1%;
67
68     may_contain = [ cntImage ];
69     embedded_obj_cnt = zipf(13);
70 };
71
72 Content cntDownload = {
73     kind = "download";
74     mime = { type = undef(); extensions = [ ".exe": 40%, ".zip", ".gz" ] };
75     obj_life_cycle = olcDownload;
76     size = logn(300KB, 300KB);
77     cachable = 95%;
78     checksum = 0.01%;
79 };
80
81 Content cntOther = {
82     kind = "other";
83     obj_life_cycle = olcOther;
84     size = logn(25KB, 10KB);
85     cachable = 72%;
86     checksum = 0.1%;
87 };
88
89 // Random content similar to request bodies generated before v3.3
90 Content cntSimpleRequest = {
91     size = unif(0KB, 8KB);
92 };
```

The phases file- phases.gp

```
1  /*
2   * Commonly used Phases
3   *
4   */
5
6  // wait for some positive activity, you may want to adjust load_factor_end
7  Phase phWait = { name = "wait"; goal.xactions = 1; log_stats = false; };
8
9  // cool down for one minute
10 Phase phCool = { name = "cool"; goal.duration = 1min; load_factor_end = 0; log_stats = false; };
```