Computer Simulation of a Web Cache Server with SES/Workbench

A Thesis

Presented to the Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Andreas Svensson

August, 2001

Members of the Supervisory Committee:

Dr. Martin Herbordt
Dr. Jaspal Subhlok
Dr. Richard Barton

Computer Simulation of a Web Cache Server with SES/Workbench

____
Andreas Svensson

Approved:

____
Chairman of the Committee
Martin Herbordt, Associate Professor,
Electrical and Computer Engineering

Committee Members:

____
Jaspal Subhlok, Associate Professor,
Computer Science

____
Richard Barton, Assistant Professor,
Electrical and Computer Engineering

____
E. J. Charlson, Associate Dean,
Cullen College of Engineering

____
Fritz Claydon, Professor and Chair,
Electrical and Computer Engineering

Acknowledgments

I would like to thank my thesis committee members Dr. Jaspal Subhlok, Dr. Richard Barton, and especially my advisor Dr. Martin Herbordt for his guidance and support. My appreciation also goes to Kevin Leigh for being a mentor, and for his invaluable advice on many aspects including web cache system platform/architecture, workload characterization and model implementation. This thesis would have taken longer to complete, if it was not for the efforts of Ali Saidi in generating the Web Polygraph simulation results and the hardware/software workload parameter measurements. My early research for this thesis was funded in part by the National Science Foundation through CAREER award #9702483 and by the Texas Advanced Technology Program under grant number 003652-0424. Last but not least, I would like to thank Gary Thome and Compaq Computer Corporation (Industry Standard Server Group, ISSG), for supporting me with equipment and funding during the last four months.

Computer Simulation of a Web Cache Server with SES/Workbench

An Abstract

of a Thesis

Presented to the Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Andreas Svensson

August, 2001

Members of the Supervisory Committee:

Dr. Martin Herbordt
Dr. Jaspal Subhlok
Dr. Richard Barton

Abstract

In web caching, frequently accessed web objects are temporarily stored in an intermediate proxy server to speed up client response time, to reduce network bandwidth usage, and to reduce server load. This thesis focuses on characterizing the internal workload of a Web cache server and in modeling an architecture to represent a physical hardware platform. The goal is to capture the most relevant parameters and functions of a Web server system with a model that can be used to characterize its behavior as well as to improve its performance. SES/Workbench™ is chosen as the simulation environment for its flexibility and richness in representing a transaction-based statistical model. Simulation results are verified against empirical data measured on a web cache hardware platform stressed by a set of Web Polygraph test clients and servers. The comparison shows that the model gives a close approximation of the workflow of a physical system. As expected, hard disks are the bottleneck for the two-drive configuration. This work shows a novel approach in modeling hardware system by using SES/Workbench™. The model can be extended to address various hardware configurations, and different network workload. Experience in capturing the parameters also highlight the fact that today's systems do not provide adequate facilities to easily acquire parameters to accurately represent the system characteristics. The model and the simulation method developed can serve as a basis for an alternate mean for architecture trade-off analysis, hardware system configuration, and performance analysis.

Table of Contents

1 Introduction ... 1
1.1 Why Web caching? ... 2
1.2 Where is Web caching performed? ... 4
1.3 Literature review ... 6
1.4 Concluding remarks ... 9
2 Networking Protocols ... 10
2.1 OSI (Open Systems Interconnection) ... 11
2.1.1 Physical layer ... 14
2.1.2 Data Link layer ... 14
2.1.3 Network layer ... 15
2.1.4 Transport layer ... 15
2.1.5 Session layer ... 16
2.1.6 Presentation layer ... 16
2.1.7 Application layer ... 16
2.2 TCP/IP ... 16
2.2.1 Transport Control Protocol (TCP) ... 18
2.2.2 Internet Protocol (IP) ... 20
2.3 Ethernet and Fast Ethernet ... 21
2.4 Request handling ... 22
2.5 Hypertext Transfer Protocol (HTTP) ... 23
2.5.1 HTTP characterizations ... 24
2.5.2 HTTP communication ... 25
2.5.3 HTTP/1.1 vs HTTP/1.0 ... 25
2.5.4 Cache-Control HTTP Headers ... 26
2.5.5 Validators ... 27
3 Present Web Cache Architecture ... 29
3.1 Data flow ... 30
3.2 Current Implementation Choices ... 32
3.3 Comparison between Web Cache and Memory Cache ... 34
3.4 How Web Caches work ... 35
3.5 Location of Web Caches ... 36
4 Model implementation ... 38
4.1 An overview of the model ... 41
4.2 The client model ... 46
4.3 The network model ... 50
4.4 The server model ... 51
4.5 The cache model ... 56
4.5.1 The networking interface ... 59
4.5.2 The CPU interface ... 60
4.5.3 The client connection interface ... 64
4.5.4 The Sending packets interface ... 67
4.5.5 The server interface ... 68
4.5.6 The *other* interface ... 69
4.5.7 The Disk access interface ... 69
4.6 Delay variables ... 71
4.6.1 CPU Thinking time ... 71
4.6.2 Data transfer ... 72
4.6.3 Other delays ... 73
4.7 Model Parameters ... 74

5 Results 78
5.1 Validation 78
5.2 Additional Experimentation 82
6 Conclusion 86
6.1 Summary 86
6.2 Model extensions 88
Appenix A: SES/Workbench 95
A.1 SES/Workbench collection 95
A.2 SES/Workbench benefits 96
A.3 System Design 96
A.4 The Animated Simulator 97
A.5 Model Components 97
A.5.1 Resource-management nodes 98
A.5.2 Transaction flow-control nodes 101
A.5.3 Submodel-management nodes 104
A.5.4 Miscellaneous nodes 105
Appenix B: The Zipf distribution 107

List of figures

Figure 1: Clients linked to a Cache server 1
Figure 2: Caching performed at the client location 5
Figure 3: Caching performed at the proxy server location (forward cache) 5
Figure 4: Caching performed at the web server location (reverse cache) 6
Figure 5: Protocol architectures 11
Figure 6: The OSI encapsulation on the sender side 12
Figure 7: OSI communication between two systems 13
Figure 8: The OSI breakdown on the receiver side 14
Figure 9: TCP/IP Protocol Architecture Model 17
Figure 10: Protocol data units in the TCP/IP architecture 18
Figure 11: TCP Header 19
Figure 12: The IP Header 20
Figure 13: The Ethernet frame 22
Figure 14: Communication process between client, cache and server 23
Figure 15: Compaq Cache Appliance hardware model 30
Figure 16: Data flow inside the web cache 31
Figure 17: Client – Network – Server model 40
Figure 18: The model with the cache included 40
Figure 19: Clients - Cache - Servers model 41
Figure 20: State machine for the client network interface 48
Figure 21: SES/Workbench model of the client 49
Figure 22: SES/Workbench model of the network 51
Figure 23: The state machine for the servers 53
Figure 24: SES/Workbench model of the server 53
Figure 25: Simplified state diagram for the cache 57
Figure 26: The networking interface 59
Figure 27: Check the buffer for data 60
Figure 28: CPU state diagram for cache dataflow activities 63
Figure 29: The CPU interface implementation in SES/Workbench 63
Figure 30: The NIC to memory transfer procedure 64
Figure 31: Client connection state diagram 65
Figure 32: State diagram for cache hit 66
Figure 33: Processing a packet from a client 67
Figure 34: The sending packets submodel 67
Figure 35: Server connection state diagram 68
Figure 36: The server connection submodel 69
Figure 37: The submodel for handling other types of tasks 69
Figure 38: The submodel for disk access 70
Figure 39: Average response time comparison 84
Figure 40: Hit response time comparison 84
Figure 41: Miss response time comparison 84
Figure 42: Request response time for the SES/Workbench model 85
Figure 43: Load balancing 89
Figure 44: Cache clustering 89
Figure 45: Submodel page nodes 98
Figure 46: Zipf distribution with linear scales and logarithmic scales on both axes 107
Figure 47: Page popularity 108

List of tables

Table 1: Typical maximum transmission units (MTUs) 17
Table 2: NLANR's Third Cache-Off cache configurations 33
Table 3: Object size distribution for Polygraph 39
Table 4: The status of the packet.id 43
Table 5: The possible values for the packet.object_type 43
Table 6: The possible values for the packet.hit_type 43
Table 7: Memory hit ratio for different memory sizes 46
Table 8 : The states of the process.connection 55
Table 9: The possible values for the cpu_packet.type 61
Table 10: CPU task priority levels 61
Table 11: First branch for CPU task 62
Table 12: Delays for CPU thinking time 71
Table 13: Data transfer delays 73
Table 14: Other delays 74
Table 15: Adjustable parameters of the model 75
Table 16: Traffic flow parameters 76
Table 17: Object parameters 77
Table 18: Other environment parameters 77
Table 19: Results from the validating simulation 81
Table 20: Results from the faster CPU simulation 83

1 Introduction

The exponential growth of the Internet has lead to a search for methods of improving performance beyond the trivial and relatively costly mechanism of adding physical connections. Web caching was introduced as an attempt to reduce response time, bandwidth consumption and server load. The response time is reduced since objects are often stored closer to the client. There is less traffic between clients and the servers, which means less bandwidth is required. The demand of bandwidth has increased with the growing use of Internet, so any solution that can reduce the traffic is welcome. If the cache can handle some of the requests, it also means that there will be less work for the server to do.

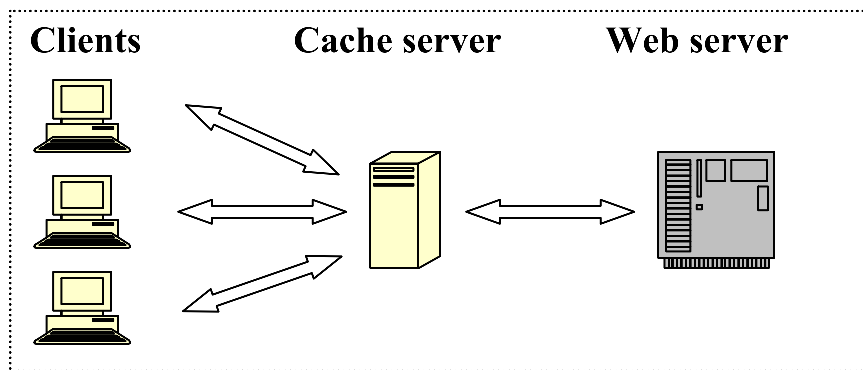


Figure 1: Clients linked to a Cache server

Most of the effort has been focused on finding optimal location of web caches, superior replacement algorithms, and efficient hierarchy design, while the architecture of the actual web cache machine has received less attention. The concept of a "web cache architecture" is mainly used for clusters of proxy server caches and how they relate to each other. On the other hand, web cache architecture can also involve designing a cache server specialized for handling HTTP (HyperText Transfer Protocol) requests and storing objects locally. Not much work has been done to improve the performance of the single web cache box. Instead of just increasing memory (RAM – Random Access Memory) and disk space, there are other things that can be done. However, due to time-to-market and cost reasons, existing hardware has been used and practically any fairly fast PC can be used as web cache, if only the memory and storage space are

large enough. Another reason for using already-existent hardware is that the market for web caches has not been big enough. It would not be very cost-efficient to spend a lot of money to design a web cache if you cannot sell enough units at the end.

A web-caching server is comparable to a normal web server, though a web-caching server does not execute CGI scripts or perform data encryption. Most traditional web-caching servers send and receive static files only. A system with several CPUs will therefore hardly offer any advantages over a system with a single CPU structure. A system with several CPUs will be useful only if the system load is high and the various processors can handle certain I/O tasks. In web caching, about 70% of disk I/O actions are write operations. The use of RAID and/or mirror solutions is therefore less useful. Disk striping and/or dividing the disks among several controllers, on the other hand, may be useful.

1.1 Why Web caching?

There are several reasons why research in the Web caching area is so important. It has been suggested that web access follows the Zipf distribution[1], which means that 10% of the web material answers for 90% of the traffic. The main goal is to store as much of these 10% at the local cache as possible. This will give several positive outcomes:

- **Reduced cost of Internet traffic.** Traffic on the web consumes more bandwidth than any other Internet service. Therefore any method that can reduce the bandwidth requirements is welcome, especially in parts of the world in which telecommunication services are expensive. Even when telecommunication costs are reasonable, a large fraction of traffic is destined to or from the U.S. and so must use expensive trans-oceanic network links.
- **Reduced latency.** One of the most common end-user desires is for more speed and many people believe that web caching can help reduce the "World Wide Wait." Latency

[1] See Appendix B

improvements are most noticeable in areas of the world in which data must travel long distances, accumulating significant latency as a result of speed-of-light constraints, accumulating processing time by many systems over many network hops, and increased likelihood of experiencing congestion as more networks are traversed to cover such distances. High latency because of speed-of-light constraints is particularly taxing in satellite communications.

Since the cost of bandwidth continues to drop, one might argue that research in Web caching is unnecessary. However, there are still many potential benefits:

- **Bandwidth will always have some cost.** The cost of bandwidth will never reach zero, although costs are currently going down as competition increases, the market grows, and economies of scale contribute. No matter what the cost for bandwidth, one will always want to maximize the return on investment, and caching will often help.
- **Non-uniform bandwidth and latencies.** Because of physical limitations such as environment and location, as well as financial limitations, there will always be variations in bandwidth and latencies. Caching can help to smooth these effects.
- **Bandwidth demands continue to increase.** New users are still being connected to the Internet in large numbers. Even as growth in the user base slows, demand for increased bandwidth will continue as high-bandwidth media such as audio and video increase in popularity [4]. If the price is low enough, demand will always outstrip supply. Additionally, as the availability of bandwidth increases, user expectations are also likely to increase.
- **Hot spots in the web will continue.** While some user demand can be predicted (such as for the latest version of a free web browser), and thus have the potential for intelligent load balancing by distributing content among many systems and networks, other popular web destinations come as a surprise, sometimes as a result of current events, but also

potentially just as a result of desirable content and word of mouth. These hot spots will continue to affect availability and response time and can be alleviated through web caching.

- **Communication vs. computation.** Communication is likely to always be more expensive (to some extent) than computation. We can build CPUs that are much faster than main memory, and so memory caches are utilized. Likewise, caches will continue to be used as computer systems and network connectivity both get faster.

There are a number of parallels to this situation; one is that of main memory in computer systems. The cost of RAM has decreased tremendously over the past decades. Yet relatively few people would claim to have enough, and in fact, demand for additional memory to handle larger applications continues unabated. Of course, there are additional secondary benefits to web caching. These include reduced load on the originating servers and improved reliability as objects may be available in a cache even when the original web server is currently inaccessible.

1.2 Where is Web caching performed?

Caching can be performed by the client application and is built in to most web browsers. There are a number of products that extend or replace the built-in caches with systems that contain larger storage, more features, or better performance. In any case, these systems cache net objects from many servers but all for a single user.

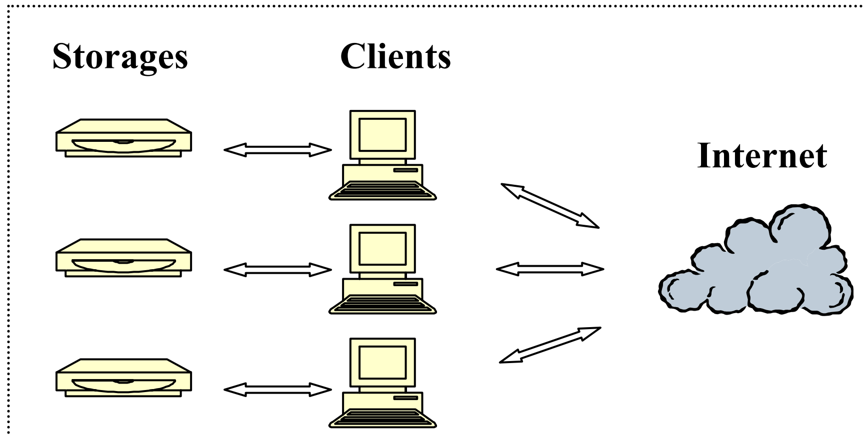


Figure 2: Caching performed at the client location

Caching can also be utilized in the middle, between the client and the server , as part of a proxy. Proxy caches are often located near network gateways to reduce the bandwidth required over expensive dedicated Internet connections. These systems serve many users (clients) with cached objects from many servers. In fact, much of the usefulness (reportedly up to 80% for some installations) is in caching objects requested by one client for later retrieval by another client. For even greater performance, many proxy caches are part of cache hierarchies, in which a cache can inquire of neighboring caches for a requested document to reduce the need to fetch the object directly.

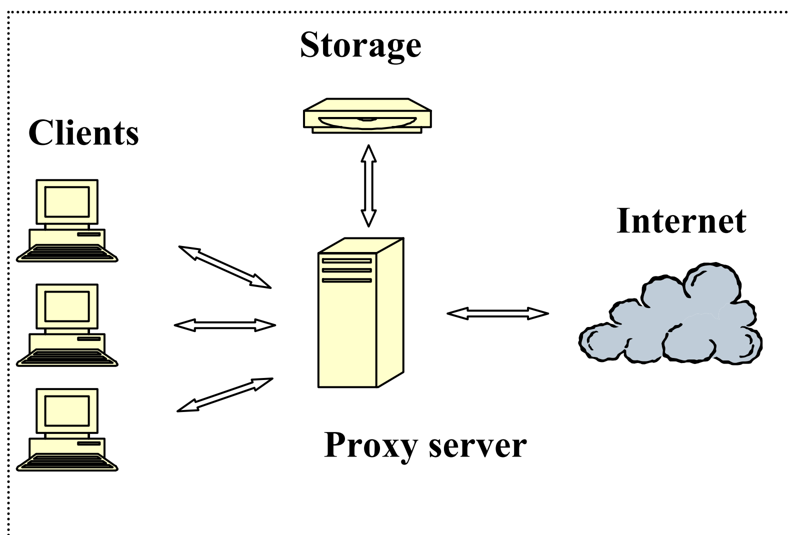


Figure 3: Caching performed at the proxy server location (forward cache)

Finally, caches can be placed directly in front of a particular server, to reduce the number of requests that the server must handle. Most proxy caches can be used in this fashion, but this form has a different name (reverse cache, inverse cache, or http accelerator) to reflect the fact that it caches objects for many clients but from (usually) only one server.

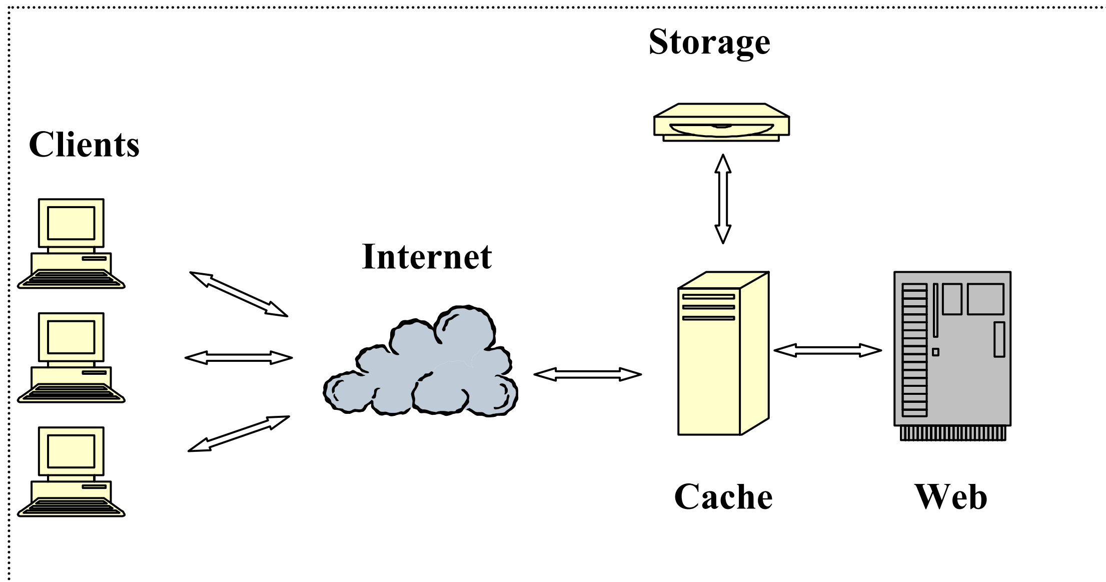


Figure 4: Caching performed at the web server location (reverse cache)

1.3 Literature review

There has been much research done in web caching, but it has mainly been focused on geographical architecture, clustering, and caching algorithms. Not many people have been looking at the actual workflow inside the web cache server. Loeb et al. [14] from IBM Corporation analyzed the PCI bus throughput when they tested a Gigabit Ethernet Adapter. They tried to maximize the amount of data a server can receive from a number of clients. They conclude that the PCI bus transfer rate is much faster than the gigabit channel transfer rate and that there is a linear relationship between server processor speed and maximum Ethernet throughput. Feldman investigated HTTP traffic and claimed that the TCP connection setup time was significant [20]. Further, she stated that the use of HTTP/1.1 versus HTTP/1.0 for requests

increased from 33% (July 98) to 71% (November 99). The usage of Get-If-Modified-Since was 17%.

There are two main types of WWW cache modeling. The first approach concerns modeling individual components such as generating representative web traffic and feeding real caches in order to test them. Well-known examples of this approach are Web Polygraph [33], the Wisconsin Proxy Benchmark [29] and SURGE [30]. The other approach consists of mathematical modeling at a higher level of abstraction where the variables of interest usually represent average values. For example, Breslau et al. [1] develop explicit formulas for the hit rate as a function of cache size of a single proxy using probability theory. Two other examples are Baentsch et al. [31] and Belchev et al. [32] where the authors describe models comprising various levels of a caching infrastructure. The advantage of the mathematical approach is that the models are relatively fast to simulate. The disadvantage is that often these models lack the desired detail. That is, oversimplifying assumptions are made which might turn out to be significant.

Baker et al. [12] used the SES/Workbench to develop a simulation appropriate for the analysis of distributed scheduling algorithms. The reason to choose SES/Workbench was that it provides mechanisms useful for the implementation of a range of scheduling policies and a simple means of altering the model parameters. Hariharan et al. [13] built a model in a distributed OO (Object-Oriented) environment to simulate end-to-end performance of web server architectures. They applied the model to Active Server Page (ASP) and Common Object Model (COM) in Microsoft's Internet Information Server and to the Java Server Page (JSP) and JavaBean technology in both IIS and Netscape Enterprise Server (NES). The results show that the ASP Script Engine performance prediction match the observed one in the test environment, but for the JSP Script Engine it predicted higher throughput at high load than the laboratory test results showed.

Bolot and Hoschka [23] studied how overloads can be avoided by carefully dimensioning the server. They also looked at performance issues related to dimensioning and caching. James

Rubarth-Lay [22] tried to optimize web performance and Maltzahn et al. [18] looked at the performance issues of enterprise-level web proxies. They found that the cache-hit ratio was around 30% and that it had little effect on the request response time. Goldberg et al. [21] also investigated cache-hit ratio by studying web proxy traces. Lee and Tomlinson [19] from the Novell Advanced Development Group present workload characteristics of a new test workload and focus on its effect on the development of technology prototypes that effectively eliminate the impedance mismatch between the disk and the rest of the proxy cache system.

Many studies have also looked at the characteristics of the web traffic [15] [16] [17], and document popularity. Crovella and Bestavros [26] studied the self-similarity of WWW traffic and the effect caching has on it. Breslau et al. suggested that web requests follow a Zipf-like distribution [1]. Woodruff et al. [25] and Mah [24] have investigated different types of documents from the World Wide Web. The latter found that the mean number of files per document is 3, but the median is 1. Mah gathered packet traces of HTTP network conversations and constructed an empirical model of HTTP traffic.

Attempts to create web caches with a specialized Web caching software integrated with the kernel, instead of running it on the general-purpose operating system, have been made [3]. A microkernel that provides the execution semantics and interfaces necessary to reduce the observed impedance mismatches between the design assumptions of general-purpose operating systems and the requirements of Internet server applications has been invented and tested. This microkernel provides large-scale resource provisioning for execution contexts, network connections, and persistent storage objects, along with innovative semantics for context scheduling, event notification, and I/O transport. The research proves that a specialized operating system/microkernel can significantly improve the Web caching performance.

Some research has also been focused on caching dynamic contents and streaming media [28] [34]. This is interesting, since increasingly companies offer services on the net with personalized web pages. It will also make it possible to store programs and other data locally.

Many people think that in the future we will be able to watch TV over the Internet. Users can already listen to radio/music, watch movie previews, and hold interactive video-conferences. The increased used of video on the Internet creates a demand of ability of caching streaming media. This is one of the areas where we will notice the greatest visual difference. Even though the networks are becoming faster, it is far from enough to allow Internet TV.

1.4 Concluding remarks

Web caching is built upon the idea that storing popular web objects locally speeds up request response time and reduces the bandwidth usage and server load. There are some particular requirements for a web cache server to run. Fast I/O (memory, disk and network) is more important for Web caching servers than fast CPUs. The system should also have sufficient memory, since swapping is fatal to its performance. Since web caching involves the transport of large amounts of data, disk I/O (bus[2] bandwidth, search and data transmission speed of the disks) is an important factor and should be proportional to the bandwidth of the network connection.

How big an impact do variations of these basic components have on the end performance? What is more important to improve to get better results? In what area should the research be concentrated and on what should we consider spending more money on for maximum improvement? These are some of the questions that will be discussed in this thesis. A model of a web cache has been built with SES/Workbench to get detailed understanding of the web cache workload and bottlenecks on traditional architecture. The results from the simulations have been compared with hardware measurements and Polygraph simulations.

The chapters are organized as follows. Chapter 2 discusses protocols and request handling. Chapter 3 describes the hardware model that will be used and Chapter 4 the experiment. Chapter

[2] Bus – One or more conductors that serve as a common connection for a related group of devices.

5 discusses the results and an explanation of the simulation tool SES/Workbench and the Zipf-distribution can be found in the Appendix.

2 Networking Protocols

To allow two computers to communicate with each other over a medium, some specific standards have to be set. Both systems need to know how to interpret the received information and what principles to follow. All the communication work is usually not done at one step, but in several layers[3]. Each layer receives bits for a higher layer, adds its header bits and sends it to a lower level. At the final stage the packet is transmitted over a medium, such as an Ethernet network, and then interpreted in the reverse order by the receiver. There are two competing sets of Internet protocols, OSI (Open Systems Interconnection) and TCP/IP. Since TCP/IP was established before

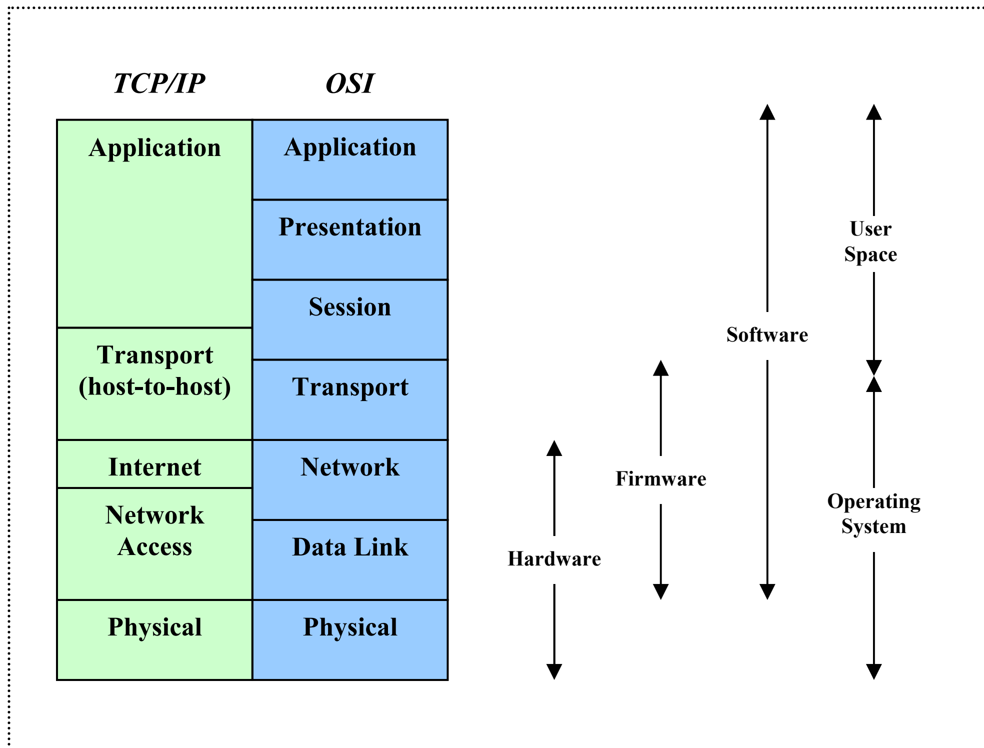


Figure 5: Protocol architectures

OSI and the delivery of the complete OSI package was delayed, TCP/IP has become the dominant commercial architecture and the protocol suite upon which the bulk of new protocols development is to be done. Important was also that the Internet was built on the foundation of the TCP/IP and the dramatic growth of the Internet, and especially the World Wide Web, cemented the victory of TCP/IP over OSI.

2.1 OSI (Open Systems Interconnection)

The OSI model is partitioned into a hierarchical set of layers. Each layer performs a related subset of the functions required to communicate with another system, relying on the next lower layer to perform more primitive functions and to conceal the details of those functions, as it provides services to the next higher layer.

An application that wants to send data to another system buffers the data at the application layer (Layer 7). That layer adds an application layer header to the Protocol Data Unit (PDU), and sends the chunk of bits to the next layer, the presentation layer. The Layer 6 work is done and the presentation header is added before it is sent to the next lower level. When the first level is reached, the physical level, the bit stream can be sent in packets to the receiver.
There is no direct communication between peer layers except at the physical layer. That is, above the physical layer, each protocol entity sends data down to the next lower layer to get the data across to its peer entity.

---

[3] Layer – A group of services, functions, and protocols that is complete from a conceptual point of view, that is one out of a set of hierarchically arranged groups, and that extends across all systems that conform to the network architecture.

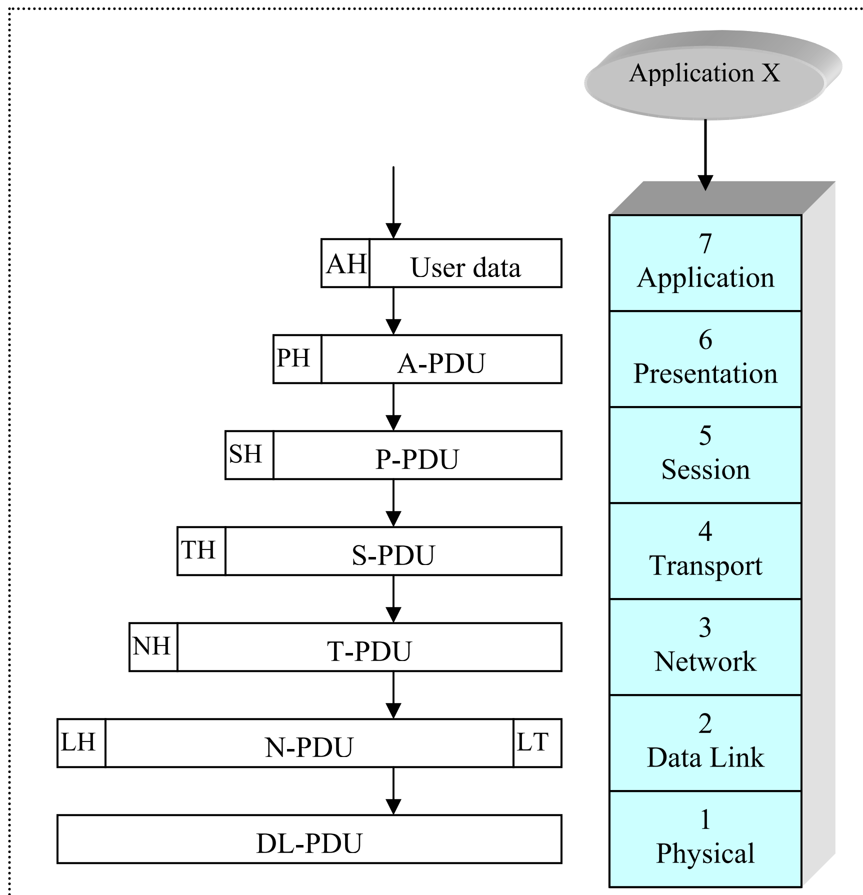


Figure 6: The OSI encapsulation on the sender side

The packets are sent over a medium, a communications network[4], such as a Wide-Area Network (WAN), Local-Area Network (LAN) or point-to-point link. The sending computer must provide the network with the address of the destination computer, so that the network can route the data to the appropriate destination. The sending computer may wish to invoke certain services, such as priority, that might be provided by the network.

[4] Communications network – A collection of interconnected functional units that provides a data communications service among stations attached to the network.

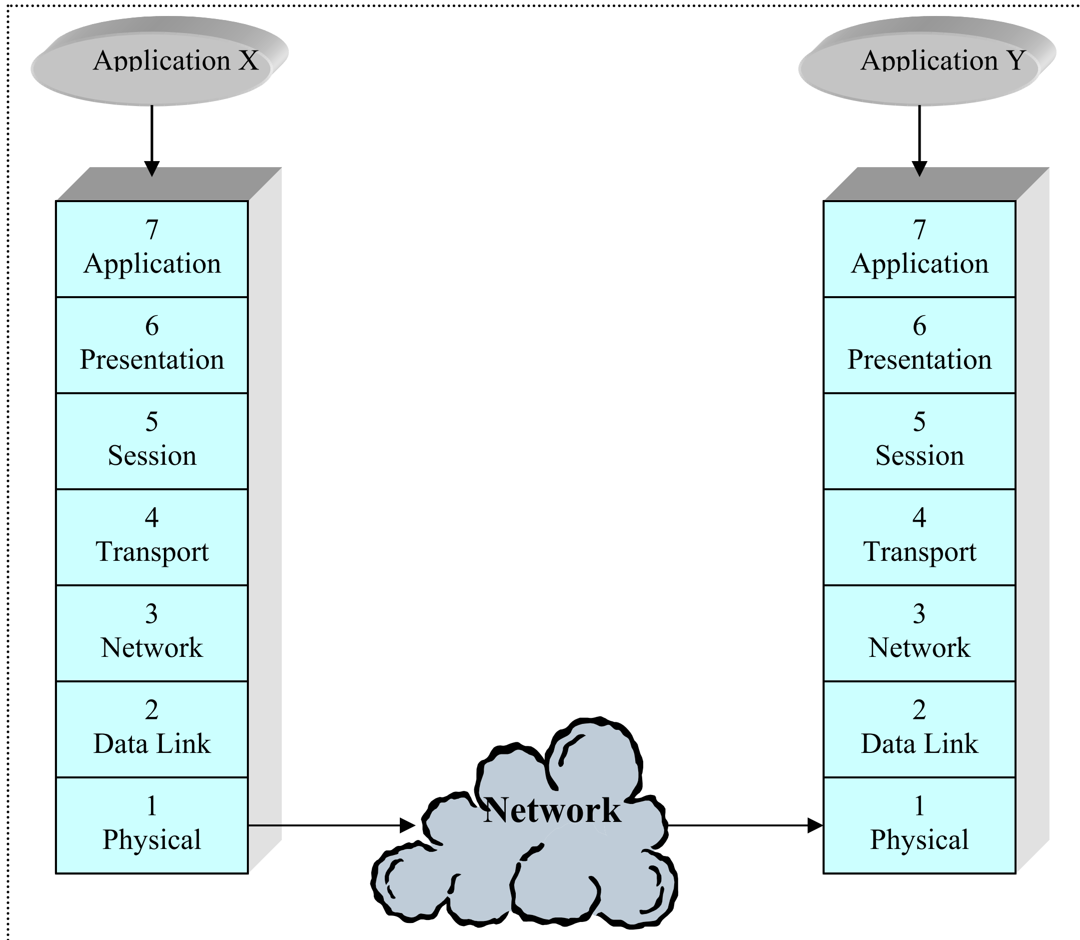


Figure 7: OSI communication between two systems

On the receiver side, the packets are broken into smaller pieces as they travel to a higher layer. The physical layer receives the packets in form of bits and does its designated work before the bits are sent to the closest higher level, the data link layer (Layer 2). The data link header and trailer are removed and processed before the bits are sent to next level. The work being done at the receiver is the reverse process of the sender. When the bits reach the application layer (Layer 7), the last header is removed and we have the desired data available. It will be sent to the application, which can use the information and create a reply. Note that the work done in the first two layers is usually done in hardware, while the rest has to be done in software (which keeps the CPU busy).

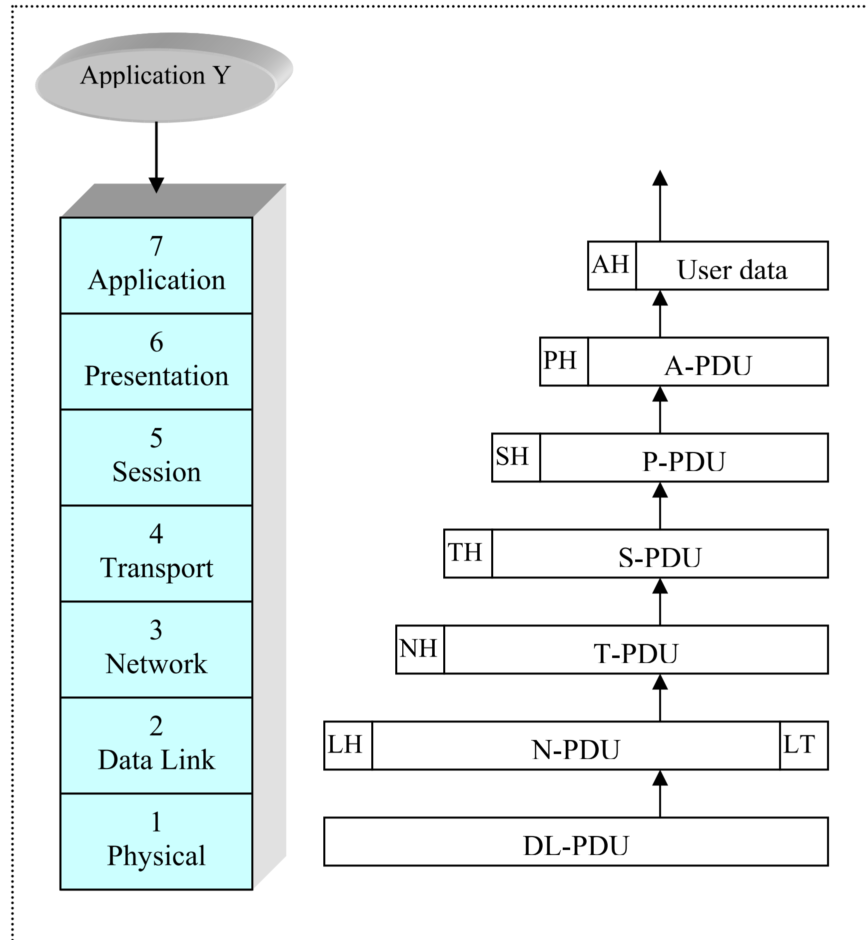


Figure 8: The OSI breakdown on the receiver side

2.1.1 Physical layer

The physical layer is the actual physical interface between devices. This is the only layer that talks directly to the corresponding layer at another device. It deals with a raw stream of bits, which are converted to electrical or optical signals and then transmitted over a circuit.

2.1.2 Data Link layer

The data link layer attempts to make the physical layer reliable while providing means to activate, maintain and deactivate the link. It mainly deals with error detection and control, which means that the next higher layer may assume error-free transmission over the link. The Media Access

Control (MAC) layer is a sub-layer of the data link layer, to deal with issues specific to a particular type of LAN. A bridge is a Layer-2 switch.

2.1.3 Network layer

The network layer provides some sort of communications network for transfer of information between end systems. It relieves higher layers of the need to know anything about the underlying data transmission and switching technology used to connect systems. There are several different communication facilities to be managed by the network layer. The system could be connected across a single network, such as circuit-switching[5] or packet-switching[6], or there could be a direct point-to-point link between two stations. In the latter case, there may be no need for the network layer, since the data link layer can perform the necessary function of managing the link. Layer 3 performs route calculation, packet fragmentation, reassembly and switching. A router is a Layer-3 switch.

2.1.4 Transport layer

The transport layer is the first layer, which handles reliable end-to-end communication. It's a mechanism for exchanging data between two systems. It handles error conditions introduced by the network such as packet lost, duplication, out of ordering, fragmentation and reassembly. TCP

---

[5] Circuit switching – A method of communicating in which a dedicated communication path is established between two devices through one or more intermediate switching nodes. Unlike packet switching, digital data are sent as a continuous stream of bits. Bandwidth is guaranteed and delay is essentially limited to propagation time. The telephone system uses circuit switching.

[6] Packet switching – A method of transmitting messages through a communication network, in which long messages are subdivided into short packets. The packets are then transmitted like message, which can take independent routes.

(Transmission Control Protocol) and UDP (User Datagram Protocol) are two common transport layer protocols.

2.1.5 Session layer

The session layer deals with dialogues and offers functions as half-duplex dialogue, checkpoints in the data-transfer stream to permit backup and recovery, and ability to interrupt and resume a dialogue and chaining. In many cases there will be little or no need for session-layer services.

2.1.6 Presentation layer

The presentation layer defines the format of the data to be exchanged between applications, so the system doesn't have to worry about bit/byte ordering, and floating-point. It also performs data compression and encryption.

2.1.7 Application layer

The application layer provides a medium for application programs to access the OSI environment. The layer contains management functions and supports protocols such as: SMTP (Simple Mail Transfer Protocol), FPT (File Transfer Protocol), TELNET and HTTP (Hypertext Transfer Protocol).

2.2 TCP/IP

The TCP/IP protocol suite provides reliable end-to-end communication between systems and has been the dominating protocol architecture since 1990. It recognizes that the task of communications is too complex and too diverse to be accomplished by a single unit. Accordingly, the task is broken up into modules or entities that may communicate with peer entities in another system. There is no official TCP/IP protocol model, but the suite can be characterized as involving five layers: application layer, transport layer, Internet layer, network access layer and physical layer.

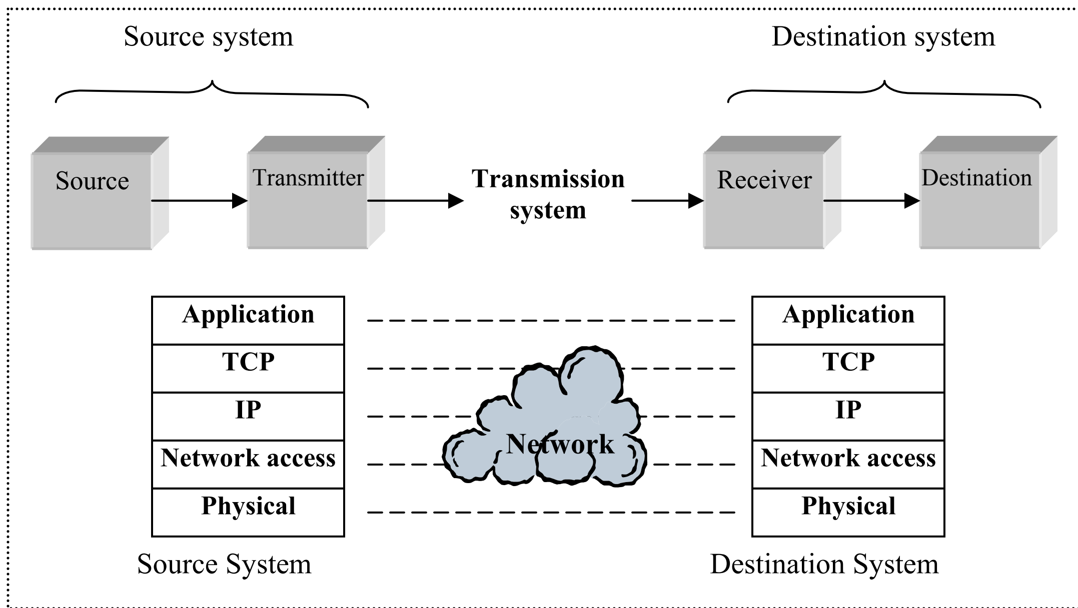


Figure 9: TCP/IP Protocol Architecture Model

As with OSI, each layer in the TCP/IP architecture adds a header to the chunk of bits, which it receives from a higher layer, before it will be send to the lower level, and eventually to the other system, via a network or other medium. The maximum transfer unit (MTU), which is the amount of data that can be included, depends on the network. If the datagram, which is to be sent, is larger than the MTU, fragmentation has to be performed. That means breaking up the datagram into smaller pieces (fragments), so that each fragment is smaller than the MTU. The MTU for Ethernet is 1500 bytes [5].

Table 1: Typical maximum transmission units (MTUs)

Network	**MTU (bytes)**
Hyperchannel	65535
16 Mbits/sec token ring (IBM)	17914
4 Mbits/sec token ring (IEEE 802.5)	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576
Point-to-point (low delay)	296

The unit of information passed by TCP to IP is called a *segment.* The IP then adds its header and sends a *datagram* to the next layer.

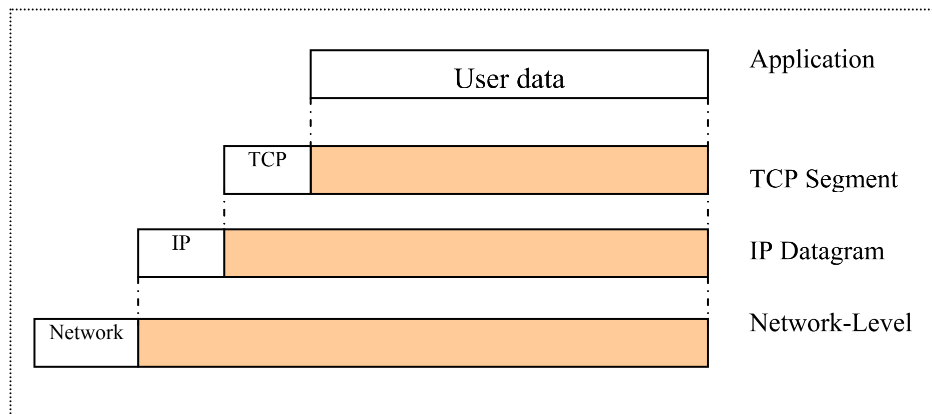


Figure 10: Protocol data units in the TCP/IP architecture

2.2.1 Transport Control Protocol (TCP)

The Transport Control Protocol (TCP) provides a connection-oriented, reliable, byte stream service and can be compared with the transport layer (Layer 4) of the OSI model. It keeps track of the blocks of data to assure that all are delivered reliably to the appropriate system. Since it is connection-oriented, the two applications using TCP (normally considered client and server) must establish a TCP connection with each other before they can exchange data.

The TCP breaks the application data into what it considers the best sized chunks to send. The maximum segment size (MSS) is the largest chunk of data that TCP will send to the other end. When the connection is established, each end announces its MSS and usually the host limits the size of the datagrams the other end sends it. This lets a host avoid fragmentation when the host is connected to a network with a small MTU. The MSS must always be at least 40 bytes less than the MTU, since the MTU includes MSS and the TCP and IP headers (20 bytes each). For IEEE 802.3 encapsulation this implies an MSS of up to 1452 bytes and for Ethernet the MSS could go up to 1460 bytes. If one end does not receive an MSS option from the other end, a default of 536 bytes is assumed. This allows for a 20-byte IP header and a 20-byte TCP header to fit into a 576-byte IP datagram.

TCP sends a segment and maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledgement isn't received in time, the segment is retransmitted. The receiver, on the other hand, receives the segment and sends the acknowledgement, with a delay of a fraction of a second. This is done in case it has some data to send in the same direction as the acknowledgement, so the acknowledgement can be sent along with it. This is sometimes called having the ACK *piggyback* with the data. Most implementations use a 200ms delay – that is, TCP with delay acknowledgement up to 200ms to see if there is data to send with it.

TCP maintains a checksum on its header and data. This is an end-to-end checksum whose purpose is to detect any modification of the data it transmits. If a segment arrives with an invalid checksum, TCP discards it and doesn't acknowledge receiving it and expects the sender to time out and resend it.

The TCP segments are being sent as IP datagrams and they can be received out of order, since IP doesn't maintain a connection between server and client. If received data is out of order, TCP has to re-sequence it before passing it in the correct order to the application. Since IP datagrams can get duplicated, TCP also has to make sure that duplicated data will be discarded.

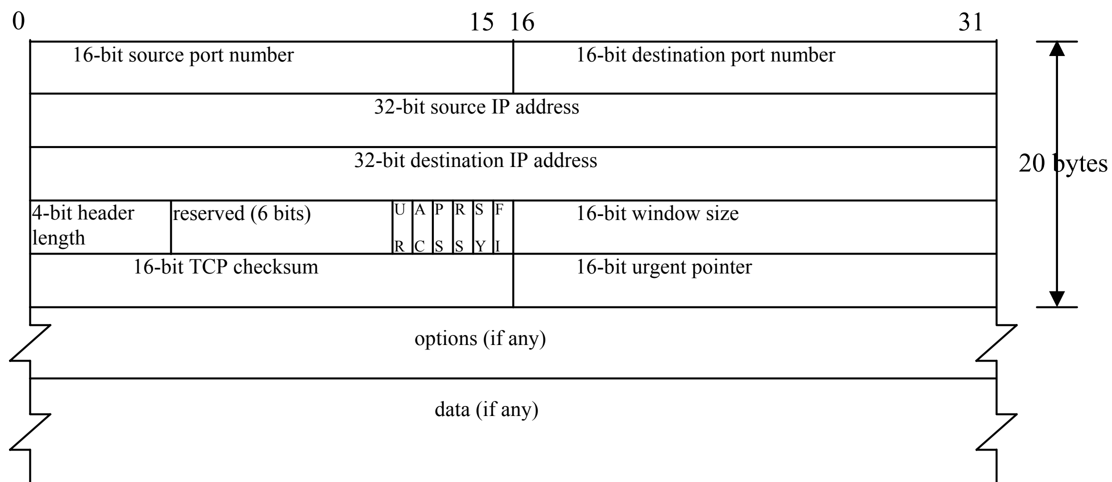


Figure 11: TCP Header

TCP also provides flow control. Each end of a TCP connection has a finite amount of buffer space. A receiving TCP only allows the other end to send as much data as the receiver has buffers for. This prevents fast hosts from taking all the buffers on a slower host.

TCP is used by many of the popular applications, such as Telnet, Rlogin, FTP, and electronic mail (SMTP).

2.2.2 Internet Protocol (IP)

The Internet Protocol (IP) is comparable to the network layer (Layer 3) in the OSI model. Here is where most of the work is being done. IP provides an unreliable, connectionless datagram delivery service and all TCP or UDP data is being transmitted as IP datagrams. The reliability has to be provided at the upper layers (e.g. TCP). IP performs fragmentation, to break up datagrams into smaller pieces, so that each fragment is smaller than the MTU. The IP header is 20 bytes and it contains the following bits:

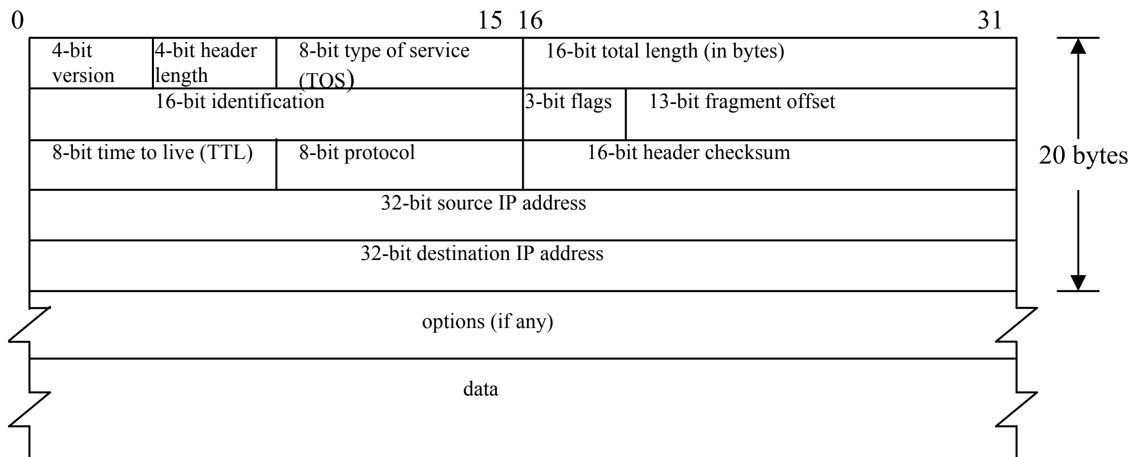


Figure 12: The IP Header

2.3 Ethernet and Fast Ethernet

The most commonly used medium access control technique for bus/tree and star topologies, is carrier-sense multiple access with collision detection (CSMA/CD). The original baseband[7] version of this technique was developed by Xerox as part of the Ethernet LAN (Local Area Network). The original broadband[8] version was developed by MITRE as part of its MITREnet LAN. All of this work formed the basis of the IEEE 802.3 standard.

The IEEE 802.3 standard includes both MAC (Medium Access Control) and CSMA/CD and the 10-Mbps specifications of it is called *Ethernet*. A faster version, IEEE 802.3 100-Mbps specifications, is called *Fast Ethernet*. Next step, the 1000-Mbps or 1-Gbps, is called *Gigabit Ethernet*.

As mentioned before, the MTU for Ethernet is 1500 bytes. That includes the headers for IP and TCP (20 bytes each), which mean that the maximum payload that can be carried by an Ethernet frame is 1460 bytes. The minimum payload is 6 bytes. When the frame is transmitted over the Ethernet physical media, 20 additional bytes have to be added, for Ethernet header and trailer. So the entire frame will be 1518 bytes. When multiple frames are transmitted, a preamble of 8 bytes and an interframe gap of 12 bytes have to be added by the physical layer. This is done in hardware and they are not included in the frame, but they will introduce wait states, which will slow down the transfer.

---

[7] Baseband – Transmission of signals without modulation. In a baseband local network, digital signals (1s and 0s) are inserted directly onto the cable as voltage pulses. The entire spectrum of the cable is consumed by the signal. This schema does not allow frequency-division multiplexing.

[8] Broadband – The use of coaxial cable for providing data transfer by means of analog (radio-frequency) signals. Digital signals are passed through a modem and transmitted over one of the frequency bands of the cable.

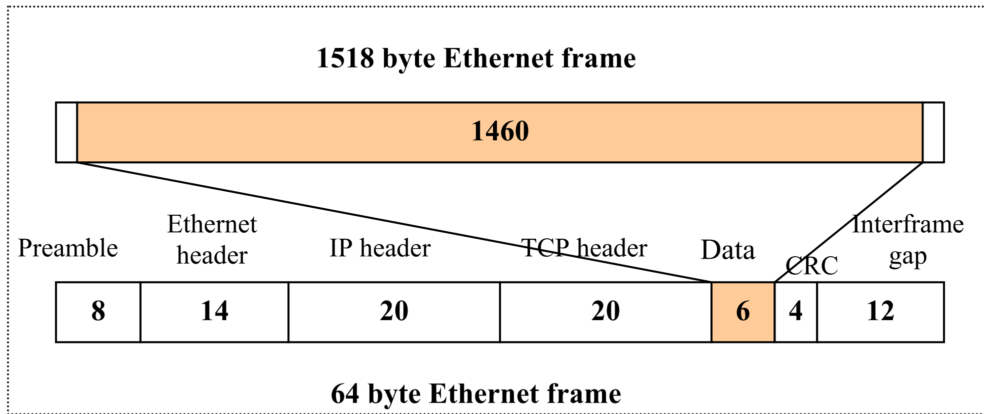


Figure 13: The Ethernet frame

According to this, the total overhead bits for multiple-frame transfer is 78 bytes. The percentage of bandwidth available varies with the Ethernet frame size from under 10 percent up to 94.9 percent (for maximum 1460 bytes payload). In case of maximum payload, the whole 1518 byte frame has to be transferred to the memory and processed by the CPU, but only the 1460 bytes will be cached.

2.4 Request handling

Before the client sends a request, a TCP connection between the client and the server (or cache in this case) has to be established. The client sends an open request and the cache replies with a synchronization packet. Then the request is sent and the cache has to find out if the object is stored locally. If it is, it checks with the server if it has been modified lately and waits for response from server. After that the data can be sent to the client, either from the cache or the server, depending on which one that has the valid object. When the transmission has been completed, the connection will be closed by the client.

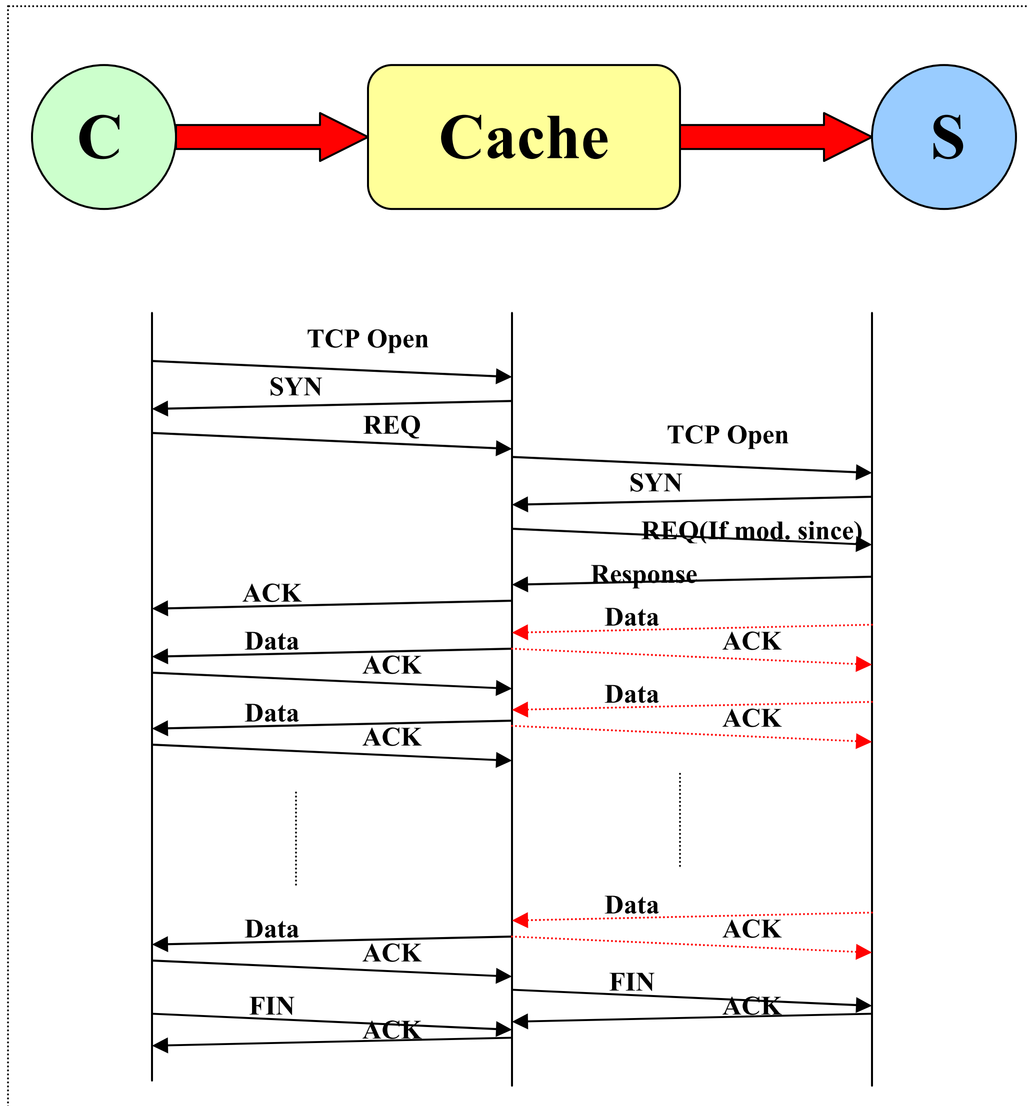


Figure 14: Communication process between client, cache and server

2.5 Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) [8] is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945 [9], improved the protocol by allowing messages to be in the format of MIME-like messages,

containing meta-information about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts. In addition, the proliferation of incompletely implemented applications calling themselves HTTP/1.0 has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities. The latest version of the protocol is referred to as HTTP/1.1and it includes more stringent requirements than HTTP/1.0 in order to ensure reliable implementation of its features.

2.5.1 HTTP characterizations

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI (Universal Resource Identifier), and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content.

A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel. A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or part of the message, and forwarding the reformatted request toward the server identified by the URI. A gateway is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol. A tunnel acts as a relay point between two connections without changing the messages. Tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary can't understand the contents of the messages.

2.5.2 HTTP communication

HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80 [10], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP presumes a reliable transport and any protocol that provides such guarantees can be used.

2.5.3 HTTP/1.1 vs HTTP/1.0

In HTTP/1.0, most implementations use a new connection for each request/response exchange. In HTTP/1.1, a connection may be used for one or more request/response exchanges, although connections may be closed for a variety of reasons.

Prior to persistent connections, a separate TCP connection was established to fetch each URL (Universal Resource Locator), increasing the load on HTTP servers and causing congestion on the Internet. The increasing use of inline images and other associated data often require a client to make multiple requests of the same server in a short amount of time. This gives significant overload and the problems with this can be solved with persistent connection, as in HTTP/1.1. Persistent HTTP connections have a number of advantages:

- By opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and memory used for TCP protocol control blocks can be saved in hosts.
- HTTP requests and responses can be pipelined on a connection. Pipelining allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time.
- Network congestion is lowered by reducing the number of packets caused by TCP opens, and by allowing TCP sufficient time to determine the congestion state of the network.

- Latency on subsequent requests is reduced since there is no time spent on TCP's connection opening handshake.
- HTTP can evolve more gracefully, since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating with an older server, it will retry with old semantics after an error is reported.

Persistent connections are good and should be used whenever possible. A client that supports persistent connections may pipeline its requests (i.e., send multiple requests without waiting for each response). A server must send its responses to those requests in the same order that the requests were received.

2.5.4 Cache-Control HTTP Headers

Although the Expires header in the HTTP/1.0 specifications is useful, it is still somewhat limited. There are many situations where content is cacheable, but the HTTP/1.0 protocol lacks methods of telling caches what it is, or how to work with it. HTTP/1.1 introduces a new class of headers, the *Cache-Control response headers*, which allow web publishers to define how pages should be handled by caches. They include directives to declare what should be cacheable, what may be stored by caches, modifications of the expiration mechanism, and revalidation and reload controls. Interesting Cache-Control response headers include:

- ***max-age=***[seconds] - specifies the maximum amount of time that an object will be considered fresh. Similar to Expires, this directive allows more flexibility. [seconds] is the number of seconds from the time of the request you wish the object to be fresh for.
- ***s-maxage=***[seconds] - similar to max-age, except that it only applies to proxy (shared) caches.

- ***public*** - marks the response as cacheable, even if it would normally be un-cacheable. For instance, if your pages are authenticated, the public directive makes them cacheable.
- ***no-cache*** - forces caches (both proxy and browser) to submit the request to the origin server for validation before releasing a cached copy, every time. This is useful for to assure that authentication is respected (in combination with public), or to maintain rigid object freshness, without sacrificing all of the benefits of caching.
- ***must-revalidate*** - tells caches that they must obey any freshness information you give them about an object. The HTTP allows caches to take liberties with the freshness of objects. By specifying this header, you're telling the cache that you want it to strictly follow your rules.
- ***proxy-revalidate*** - similar to must-revalidate, except that it only applies to proxy caches.

For example:

```
Cache-Control: max-age=3600, must-revalidate
```

2.5.5 Validators

Validators are significant; if one isn't present, and there isn't any freshness information (Expires or Cache-Control) available, most caches will not store an object at all. The most common validator is the time that the document last changed, the *Last-Modified* time. When a cache has an object stored that includes a Last-Modified header, it can use it to ask the server if the object has changed since the last time it was seen, with an *If-Modified-Since* request. HTTP/1.1 introduced a new kind of validator called the ETag. ETags are unique identifiers that are generated by the server and changed every time the object does. Because the server controls how the ETag is generated, caches can be surer that if the ETag matches when they make an If-None-Match

request, the object really is the same. Almost all caches use Last-Modified times in determining if an object is fresh, but as more HTTP/1.1 caches come online, Etag headers will also be used.

3 Current Web Cache Architecture

When designing modern web caches, people have been using traditional server architecture with extended memory and storage space. To get more understanding of the meaning of this, we should look at the model. In this study, the DL320 platform has been chosen. The CPU is a Pentium III, which runs at 933 MHz and is connected to the North Bridge through a 32-bit 133 MHz bus. The North Bridge also connects the memory, the magnetic disks, and the network interface. The memory has a 64-bit 133 MHz bus, but the bandwidth can easily be increased. The PCI[9] bus for the NIC(s) is 64 bits wide and runs at 33 MHz. Several NICs can be used, but one might be enough to handle both incoming and outgoing traffic. The network connection speed is 100 Mbps (Fast Ethernet, 100 Mega bits per second or 12.5 MB/sec).

The PCI bus, which connects the SCSI to the storage disks, is 32 bits wide and runs at 33 MHz. It means that theoretically 132 MB of data can pass per second (33M*32/8), but technically the maximum rate is usually around 100 MB/second. On the network side, the PCI bus is twice as fast, which implies that it should be able to handle several NICs, since they are more than 10 times as slow. The SCSI is a WIDE-ULTRA2, which can handle 160 MB/second. Next step, the WIDE-ULTRA3, can go up to 320 MB/second. The standard speed is 40 MB/second.

All these components can be switched as and tested, to find an optimal web cache. By using already-existing components, there are many different configurations that can be tested and evaluated with their cost taken into account. What configuration will give best Web caching performance and to what cost? What will the difference be if one component is changed and will the bottleneck shift? These are some of the question one might ask oneself when designing a web cache. This thesis focuses on this type of questions and instead of buying components and implementing them in hardware, a parameterized model was built and simulated with

---

[9] PCI – Peripheral Component Interconnect

SES/Workbench. The system type was Sun Enterprise 450 (4 x UltraSPARC-II 296MHz). The Operating system was Solaris 2.6 and the memory was 2.0GB. The following part explains the present web cache architecture and the workflow inside it.

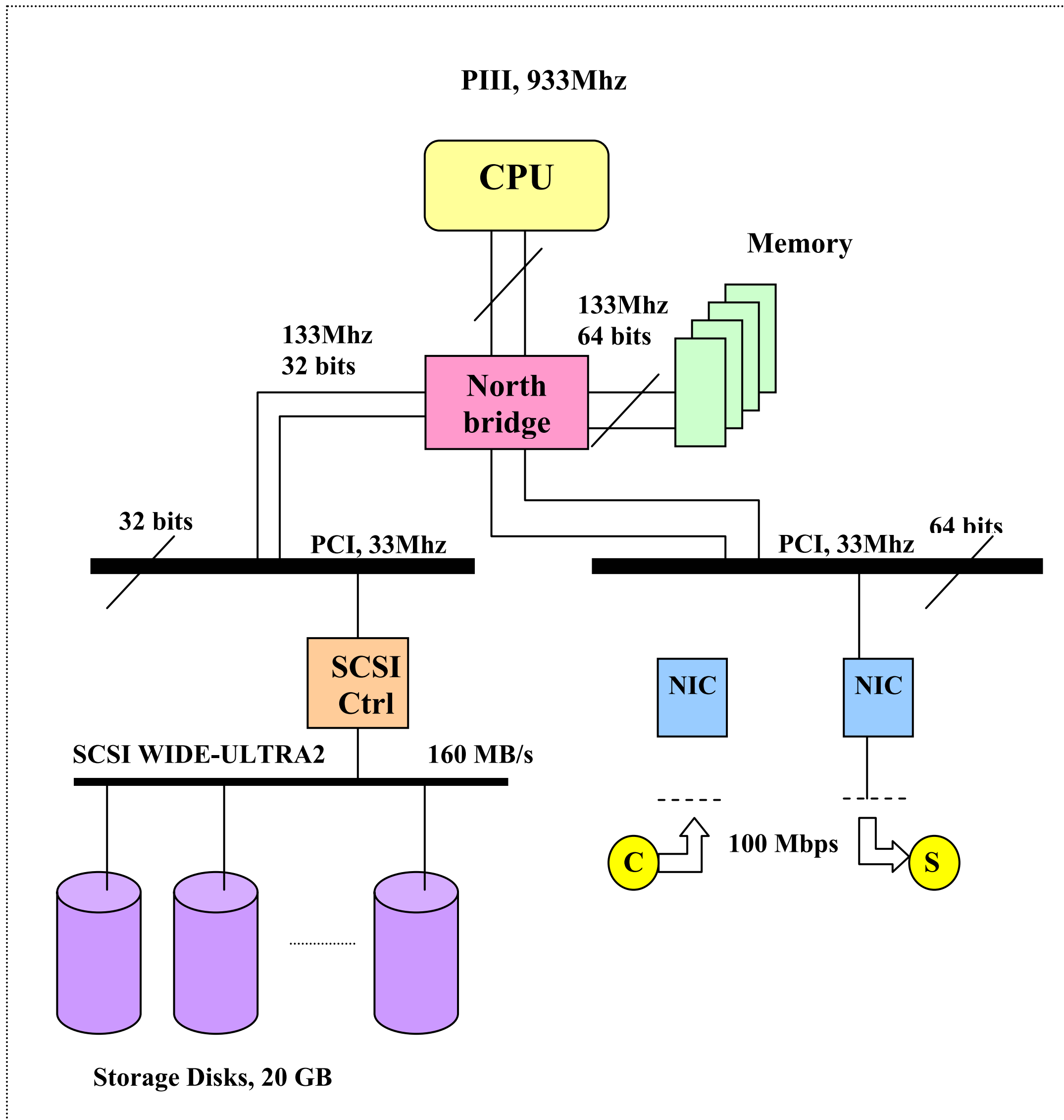
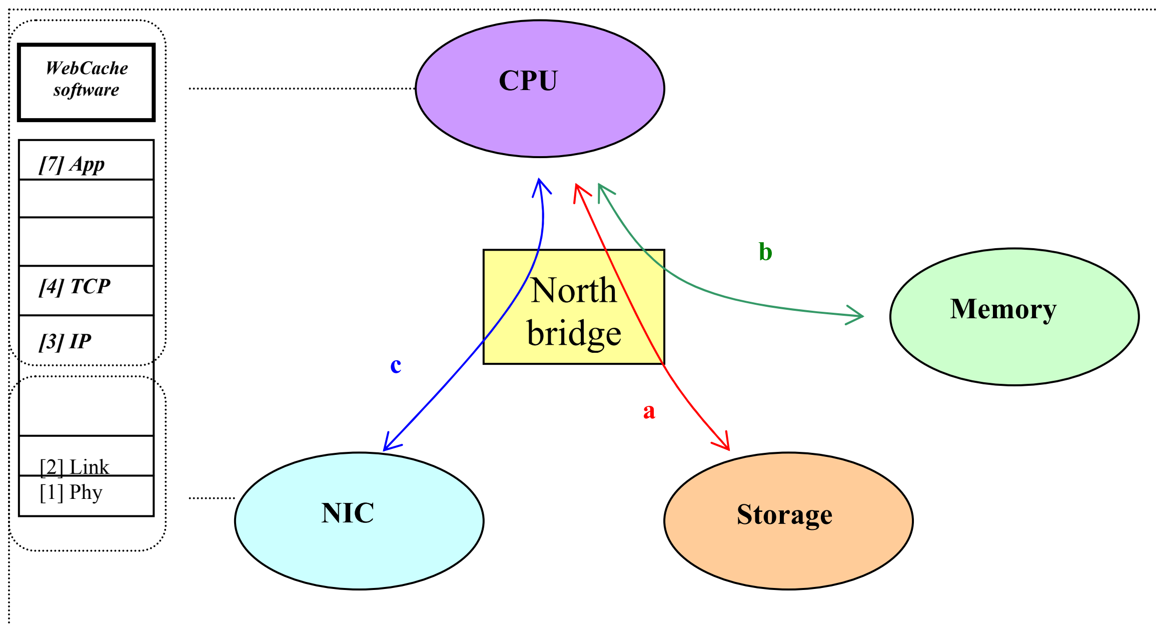


Figure 15: Compaq Cache Appliance hardware model

3.1 Data flow

A request from a client arrives at the NIC, which transforms the signals into a chunk of bits and makes the necessary data link control. To find out if this is a HTTP request, or even a packet meant for this machine, software has to process this chunk of bits, which takes CPU time. It also

requires bandwidth, since the chunk has to be transmitted to the CPU, following path c). The CPU may have to access the memory, to run OS and Web caching Application, which requires more CPU time. If it is a HTTP request, it has to check if the object is stored at this location. It can be either in the memory, or in the magnetic disks (or both). A directory of all stored objects on this machine is available in the memory. Information will flow along path b). If the object is found there, the CPU will transmit it to the NIC and send the reply to the client. That means the information has to first flow along path b) and then along path c). If the object is stored in the disks, data has to flow along path a), before it can be transmitted to the NIC and then on to the client. If it's a total miss, the chunk of bits will be sent to the NIC, a packet will be put together



again, and the request will be forwarded to the web server.

Figure 16: Data flow inside the web cache

When the server responds to the cache, another chunk of bits will reach the NIC. Again the CPU has to decode this to find out that it's a HTTP response. Now the object has to be stored in the memory (if it isn't already there) and be sent back to the client. This means that every packet has to be decoded at the NIC, transmitted to the CPU, interpreted and then send both along path b) and c), by the CPU.

Along with all this information flow inside the server, there is one more thing the Web caching software has to do. When the memory is full, objects have to be swapped out and stored at the magnetic disks. This is also handled by the CPU, so information will flow through paths b) and a).

3.2 Current Implementation Choices

Web cache server market is relatively new, about three years. For several reasons, almost all the web cache servers in the market are based on widely available generic server platform. Most notable reasons are time-to-market, development cost, and lack of economy of scale (i.e., low unit volume production). Table 2 shows a number of different existing Web caches that competed in the 3rd Cache Off event, openly held by NLANR (National Laboratory of Applied Network Research) IrCache group [6].

A typical Web cache system listed in Table 2 is configured with the following features: One or more CPU; A multi-bank central memory 256MB-4GB; Zero or more PCI I/O slots populated with Ethernet network and storage cards; and tTo or more hard disk drives (mostly SCSI).

Table 2: NLANR's Third Cache-Off cache configurations

Label	Full product name	Price (US$)	Cache units	CPUs (n · MHz)	RAM (MB)	Disks (n · GB)	NICs (n · Mbps)	Rack Space	Cache (GB)	Software
Aratech-2000	Aratech Jaguar2000	13,150	1	1 · 800	1024	8 · 18	1 · 100	n/m	132	FreeBSD 4.1
CinTel-iCache	CinTel PacketCruz iCache	9,995	1	1 · 650	512	4 · 30	1 · 100	2	116	FreeBSD 4.1, iMimic DataReactor Core v2.0b
Compaq-b17	Compaq TaskSmart C-Series b17	n/a	1	1 · 800	512	2 · 18	2 · 100	1	24	ICS 1.2.94
Compaq-C2500	Compaq TaskSmart C2500-2	61,995	1	1 · 1000	4096	23 · 09	2 · 1000	9	189	ICS 1.2.76
Dell-100	Dell PowerApp.cache 100	2,902	1	1 · 650	256	2 · 09	1 · 100	1	13	ICS 1.2.94L
Dell-200x4	Dell PowerApp.cache 200 Load Balanced Cache Array	50,876	4	4 · 866	4096	16 · 18	12 · 100	10	236	ICS 1.2.94L
F5-EDGE-FX	F5 Networks EDGE-FX Cache 1.0	9,900	1	1 · 550	512	4 · 30	1 · 100	2	114	FreeBSD 4.1, iMimic DataReactor Core v2.0b
IBM-220-1	IBM eServer xSeries 220 #1	4,108	1	1 · 800	389	2 · 18	1 · 100	n/m	30	ICS 1.2.94
IBM-220-2	IBM eServer xSeries 220 #2	5,856	1	1 · 866	512	3 · 09	1 · 100	n/m	22	ICS 1.2.94
IBM-230	IBM eServer xSeries 230	16,998	1	1 · 1000	1024	6 · 09	1 · 1000	5	42	ICS 1.2.94
IBM-330	IBM eServer xSeries 330	7,128	1	1 · 733	256	2 · 09	2 · 100	1	13	ICS 1.2.94
iMimic-1300	iMimic DataReactor 1300	6,395	1	1 · 667	512	3 · 46	2 · 100	1	132	FreeBSD 4.1, DataReactor Core v2.0b
iMimic-2400	iMimic PenguinCache 2400	9,995	1	1 · 866	512	4 · 18	1 · 100	2	70	RedHat Linux 6.2, DataReactor Core v2.0b
iMimic-2600	iMimic DataReactor 2600	18,995	1	1 · 866	1024	6 · 36	1 · 1000	2	210	FreeBSD 4.1, DataReactor Core v2.0b
iMimic-Alpha	iMimic/Alpha Content Accelerator	29,995	1	1 · 833	2048	9 · 36	1 · 1000	n/m	315	FreeBSD 4.1, DataReactor Core v2.0b
Lucent-50	Lucent imminet WebCache 50	9,950	1	1 · 600	512	2 · 20	1 · 100	2	36	FreeBSD 4.1, proprietary
Lucent-100	Lucent imminet WebCache 100	14,950	1	1 · 700	768	3 · 18	1 · 100	2	52	FreeBSD 4.1, proprietary
Lucent-100z	Lucent imminet WebCache 100z	19,950	1	1 · 800	1024	5 · 18	1 · 100	2	88	FreeBSD 4.1, proprietary
Microbits-C	Microbits Business (C-2-H)	5,430	1	1 · 600	320	1 · 09 1 · 18	1 · 100	n/m	23	ICS 1.2.94.04
Microbits-P	Microbits Pizza Box (P-1-E)	2,150	1	1 · 266	128	1 · 06	1 · 100	n/m	5	ICS 1.2.94.04
Microsoft-1	Microsoft Internet Security and Acceleration Server #1	47,991	1	4 · 700	4096	15 · 18 1 · 09	1 · 1000 1 · 1000	n/m	260	Windows 2000 Server
Microsoft-2	Microsoft Internet Security and Acceleration Server #2	4,807	1	1 · 667	384	2 · 20 2 · 18	1 · 100	n/m	56	Windows 2000 Server
NAIST-1	NAIST Kotetu v1.0 #1	4,474	1	1 · 733	1024	2 · 36	1 · 100	n/m	9	RedHat 6.2, Kotetu v1.0
NAIST-2	NAIST Kotetu v1.0 #2	13,833	1	2 · 866	1024	6 · 36	1 · 1000	6	8	RedHat 6.2, Kotetu v1.0
NetApp-C1105	NetApp C1105	10,950	1	1 · 433	512	2 · 36	2 · 100	1	64	NetCache 5.0
NetApp-C6100	NetApp C6100	100,500	1	1 · 733	3072	14 · 18	1 · 1000	11	226	NetCache 5.0
Squid-2.4.D4	Squid Version 2.4.DEVEL4	4,008	1	2 · 550	512	6 · 08	1 · 100	n/m	24	FreeBSD 4.1, Squid-2.4.DEVEL4
Stratacache-D	Stratacache Dart D-20	699	1	1 · 200	256	1 · 20	1 · 100	n/m	18	ICS 1.2.94 Micro Edtn
Stratacache-E	Stratacache Express E-55	3,295	1	1 · 500	256	1 · 18	1 · 100	n/m	16	ICS 1.2.94
Stratacache-F	Stratacache Flyer F-110	6,995	1	1 · 650	512	2 · 18	1 · 100	1	33	ICS 1.2.94
Swell-1450	Swell CPX 1450	2,679	1	1 · 800	512	3 · 15	1 · 100	2	12	Linux 2.4.0 test8

3.3 Comparison between Web Cache and Memory Cache

The word "cache" comes from the French word *cacher*, which means to press together or hide. In computer association, it means "a computer memory with very short access time used for storage of frequently used instructions or data"[10]. The cache memory is found in all today's computer architecture. A web cache is similar to the cache memory inside the computer. Web caching is the temporary storage of web objects (such as HTML documents) for later retrieval. Web clients request documents from web servers, either directly or through a web cache server or a proxy. A web cache server has the same functionality as a web server, when seen from the client and the same functionality as a client when seen from a web server. The primary function of a web cache server is to store web documents close to the user, to avoid pulling the same document several times over the same connection, reduce download time and create less load on remote servers. There are three significant advantages to Web caching:

- Reduced bandwidth consumption (fewer requests and responses that need to go over the network)
- Reduced server load (fewer requests for a server to handle)
- Reduced latency (since responses for cached requests are available immediately closer to the client being served).

Together, they make the web less expensive and better performing.

In some ways Web caching can be compared to memory caching. In that case, the memory cache would correspond to the web cache server and the PC to the world, or at least the Internet. But there are also differences between memory caching and Web caching, especially when it comes to algorithm implementations. Within the computer, the memory cache contains blocks of the same size (e.g., 32 bytes). The objects are in bits and have no meanings to the CPU. There are

---

[10] Merriam-Webster's online dictionary

some status bits that the CPU can modify, but coherence has to be maintained and corruption avoided. Use of either invalid or corrupted data can give catastrophic consequences. The web cache, on the other hand, stores objects of different kinds and varying sizes. Data integrity is desired but not critical for a web cache. If data is invalid or corrupted, a new copy can be obtained from the server, with some performance penalty. Because of this, clustering of web Caches is also easier.

3.4 How Web Caches Work

All caches have a set of rules that they use to determine when to serve an object from the cache, if it's available. Some of these rules are set in the protocols (HTTP 1.0 and 1.1), and some are set by the administrator of the cache (either the user of the browser cache, or the proxy administrator). Generally speaking, these are the most common rules that are followed for a particular request:

1. If the object's headers tell the cache not to keep the object, it won't. Also, if no validator[11] is present, most caches will mark the object as un-cacheable.
2. If the object is authenticated or secure, it won't be cached.
3. A cached object is considered *fresh* (that is, able to be sent to a client without checking with the origin server) if:
   - It has an expiry time or other age-controlling directive set, and is still within the fresh period.
   - If a browser cache has already seen the object, and has been set to check once a session.
   - If a proxy cache has seen the object recently, and it was modified relatively long ago.

[11] See Chapter 2.5.5

Fresh documents are served directly from the cache, without checking with the origin server.

4. If an object is stale, the origin server will be asked to *validate* the object, or tell the cache whether the copy that it has is still good.

Together, freshness and validation are the most important ways that a cache works with content. A fresh object will be available instantly from the cache, while a validated object will avoid sending the entire object over again if it hasn't changed.

3.5 Location of Web Caches

When it comes to location of a cache, there has been much research done. Most data have been collected by measuring the traffic at proxy servers and compared with results from different cache configurations and clusters. It has been found that organization memberships are significant [4]. Members of an organization are more likely to request the same documents than a set of clients of the same size chosen at random from all the clients in the population. It has also been suggested that the most popular documents are popular both within organizations and globally. This is something one has to think about when choosing location for the web cache. In general there are four things one should have in mind, when considering the location of a web cache:

1. Place web cache servers on the users' side of a bottleneck
2. Place web cache servers close to the flow of traffic
3. Place a local web cache server close to your Internet connection
4. Place web cache servers where more networks join together

Web cache servers should be placed as closely to the actual flow of web traffic as possible. Web traffic flows from server to client and the first-level cache should be placed close to the client. Upper level caches should be placed upstream and the top-level cache for a specific server should be placed as far upstream as possible. First level caches is typically placed on campus, close to

the institutions Internet connection. Next level caches are placed at the regional level. Each level of caching adds latency in the case of a cache miss, and more than 3 level of caches should be avoided.

More caches may co-operate to spread load and increase hit levels as well as redundancy. Models of co-operation are largely expanded in HTTP/1.1 with support for multilevel caching, with a number of new caching headers coming into use.

4 Model implementation

The proxy cache is, in most cases, based on existing hardware design to save time and money. It wouldn't be cost-efficient to spend millions of dollars to invent a new architecture just for web caches, since the market has been limited. Vendors have been using already existent components in an already existent design and simply tried to increase memory and storage space to improve the caching functionality. These are not the only factors, which decide the performance of a web cache. The CPU utilization, the speed of the PCI bus and the NIC performance will also have a significant influence.

To see what influences different hardware configurations have on the caching performance, one can get different components and put them together and run experiments with them. Another approach is to use a software simulation program and build models, which can be simulated. SES/Workbench is such a simulation tool, based queuing theory, which offers a wide range of tools, such as animated run, parameterized models and graphical user interface.

There are several things that can be alternated, e.g.:

- Processor speed
- Memory size
- Bus speed
- Storage space
- Number and speed of NICs
- Maximum Transfer Unit size (MTU)
- Request rate

A basic key is that the network runs well and that it gives a realistic workload. It has to be set up properly and there must be a way to adjust it for different environments. Once the network connection is running, variables such as transfer timeout, packet-drop rate, window size etc. don't have to be changed. The MTU can be highest possible (1500 bytes for Ethernet) and the request

rate can be simulated by having several clients requesting objects with a certain delay. If the memory is too small, it will start swapping out the cache application and the operating system when it gets full. That will give some extra delay for the processing time. To simplify the model, one can make sure the memory is big enough to avoid swapping, i.e. assume that it will never happen.

The model could be parameterized, so adjustments and reconfigurations can be done easily. Delay values, hit ratios and distributions can be taken from Polygraph experiments and other resources. If an object to be transmitted doesn't fit in one packet, it is divided into as many packets that are needed, calculated with highest possible MSS, and a certain delay is added.

Several studies have suggested that web requests follow the Zipf distribution. Other claims it doesn't. One study [1] found that the distribution doesn't follow Zipf's law precisely, but instead follows a **Zipf-like distribution** with the exponent varying from trace to trace. The hit ratio follows log (R), where R is number of request. When R gets large, the hit ratio stabilizes at a constant rate. A stabilized hit ratio of 58% for Polygraph as an upper limit has been suggested [6]. The model is called an *independent reference model* [2], where no correlation between request frequency, response size, and rate of change is assumed. Studies have showed that the correlation is week if even existing [1].

The sizes of the web objects vary a lot and their distributions depend on the type of object. The following table is object data for PolyMix-3 and can be found in [6]. The probabilities and random distributions can easily be implemented in SES/Workbench.

Table 3: Object size distribution for Polygraph

Type	Percentage	Reply Size	Cachability	Expiration
Image	65.0 %	exp(4.5kB)	80%	logn (30day,7day)
HTML	15.0 %	exp(8.5kB)	90%	logn(7day,1day)
Download	0.5 %	logn(300kB,300kB)	95%	logn(0.5year,30day)
Other	19.5 %	logn(25kB,10kB)	72%	unif(1day,1year)

The PCI busses can be tried out with both 32- and 64-bits; the bus between CPU and memory with 64- and 128-bits. The frequency can be either 33 or 66 MHz. Several NICs can be added on the PCI bus and they can be either 100-Mbps or 1-Gbps. The SCSI can have different speed, as well as the hard drives. The sizes of the hard drives and the memory can also be altered. Finally the CPU processing speed can be changed, and even the number of processors. For this experiment, there will only be a single processor, but additional processors can be added later, for further experiments. For comparison reasons, a certain request rate can be selected and conclusions can be drawn from the results.

The first step is to build a model of the network. The simplest way will be to have a client and a server, connected by a physical network. The model can be justified and when the communication between them works well, the cache can be added.

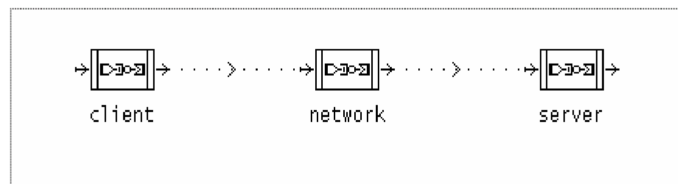


Figure 17: Client – Network – Server model

The network will be treated as a resource[12], which means the client and server will put messages (packets) in their mailboxes and the network will transport them back and forth, with a certain delay. The network will be trimmed by alternating parameters such as TCP window size, timers, latency, packet drop, and other constraints.

Figure 18: The model with the cache included

[12] See Appendix A – SES/Workbench

The two networks can have different parameters, depending on where the cache is located. For instance, if the cache is closer to the client, the network between the client and cache will be faster than the network between the cache and the server.

4.1 An Overview of the Model

The model should be seen as a client-server model, with a cache between. The same model can be implemented in other areas as well, with some simple modifications, e.g. network storage. Five submodels are linked together by a visible pointer. The simulation starts at the client side, where the client sends requests to the server. The client submodel is an array of clients and the clients can only have one connection each open at the time. The clients can therefore be seen as connections. In practice, a client can open multiple connections for a request, but for the cache it doesn't matter if there are multiple connections from one client, or just multiple clients with one connection each. The latter case will be much easier to implement in the model.

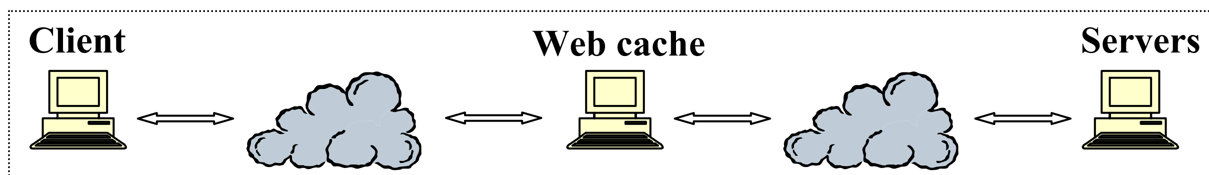


Figure 19: Clients - Cache - Servers model

Between the client and the server there is a submodel called cache. This is the actual cache server at the proxy server level. It really doesn't matter if the cache is seen as proxy level cache or server side cache. The main difference is the latency on the network and that can be modified. Another difference, that won't be studied in detailed in this case, is the type of workload. For a forward cache (close to the clients), there will be request from a limited number of clients and they will access virtually unlimited amount of data. Therefore it will be impossible to store all data in memory, which means disk access is unavoidable. For the reverse cache (close to the server), there will be accesses from virtually unlimited number of clients, but they will only access a

limited amount of data. Therefore attempts to fit all the data in the memory can be a good approach. The research of the impact of this is out of scope of this thesis.

Between the client and the cache there has to be a medium to transmit messages or packets. The submodels *link_client_cache* and *link_cache_server* are two implementations of network models, which simply forward messages to and from mailboxes. They have a certain transmitting bandwidth and latency, and all the parameters can be changed for different implementations.

The server submodel is also an array of submodels, but in contrast to the cache it can handle multiple connections. Again it doesn't matter to the server if several connections are from the same client. Each connection is still handled exclusively and it has a process with certain parameters tied to it. The client id is used to distinguish the different connections from each other.

The packets that traverse over the network have to hold some information. In the model this is implemented as variables. Instead of having source and destination address, the packet carries information about the client and the server. This means that the network and the cache need to remember where the packet came from, to find out where to forward it. The packet also has an id, which tells what kind of packet it is and what to do with it. It also has to have a packet size, which is given in bytes. Since a client can send multiple requests on one connection, each request has to have an id. The packet also has to carry information about the actual object – what type it is, where it was found, and the size. They are not used for every single packet, but they have to be set to something. SES/Workbench sets all undefined integers to 0.

```
unshared struct{
	unsigned short int client;	/* The id of the client */
	unsigned short int server;	/* The id of the server */
	int id;				/* Packet id, what kind of data */
	unsigned short int bytes;	/* Size of packet */
	unsigned char request;		/* The request number */
	char object_type;		/* The type of object */
	char hit_type;			/* Where the object was found */
	int info;			/* Additional information (object size) */
} packet;
```

Table 4: The status of the packet.id

packet.id	Client	Server
-1	TCP-open	SYN
-2	Request	ACK
-3	FIN	ACK
-4	N/A	Freshness
>0	ACK	Data

Table 5: The possible values for the packet.object_type

packet.object_type	
1	IMAGE
2	HTML
3	DOWNLOAD
4	OTHER

Table 6: The possible values for the packet.hit_type

packet.hit_type	
6	HIT
7	MISS
8	DISK
9	FRESH

If the *packet.hit_type* is FRESH, it means the object is in cache, but there is a possibility it's stale. The cache will verify with the server, which either changes it to HIT or MISS. The cache can then keep it as hit, or change it to DISK, which means that it was a hit, but the object is in disk.

Most of the network workload in the model corresponds to the characteristics from the PolyMix-3 distribution, and similar features have been implemented. Some of them are as follows:

- a mixture of content types
- persistent connections
- network packet loss

- reply sizes
- server-side latencies
- a mixture of cache hits and cache misses
- a mixture of cachable and uncachable responses
- request rates and interarrival times
- cache validation (IMS requests)

The comparison of the results with Polygraph simulations will be easier if the models are built upon the same assumptions. Absent from the cache-off workload for both PolyMix-3 and the SES/Workbench model are the following features:

- DNS-lookup latencies
- aborted requests
- HTTP content (HTML, images, etc.)
- client-side latencies, bandwidth limits
- non-HTTP traffic
- different popularity characteristics among servers

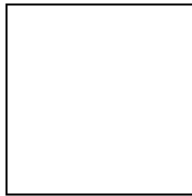
They can be implemented in the future if desired. Polygraph servers are configured with 40 millisecond delays (per packet, incoming and outgoing), and with a 0.05% probability of dropping a packet. Server think times are normally distributed with a 2.5 second mean and a 1 second standard deviation. Note that the server think time does not depend on the object type or object size. Instead, it is randomly chosen for every request. For freshness tests, the server think

time is faster if the object is fresh, since the server doesn't have to do any processing with the object.

About 20% of PolyMix-3 requests contain an "If-Modified-Since" HTTP header. In fact, the percentage of "304 Not-Modified'" responses from an ideal cache is only around 5%, far less than 20% of IMS requests. In general when a cache receives a request it checks in the memory if the object is cached. The index is stored in the memory and it holds information about where the object is cached - in memory or in disk. Most objects also have an expiration date, i.e., how long the object is fresh. The cache compares this filed with the current time and if the expiration date is in the past, or within a user-defined threshold, it has to check with the server if the object has been updated. The cache sends and Get-If-Not-Modified request to the server. The server either responds with 304-Not-Modified, or 200-Ok followed by the object data. In the SES/Workbench model, if an object is a hit, the cache checks if the object still is fresh. This takes some processing time and the decision is implemented by a random distribution. If the object is not fresh, the cache checks with the server if the object has been updated. If so, the server replies and sends the updated objects. If not, the cache gets just the reply and can send the cached object to the client. For the simulation implementation, 20% of the hits have to go to the server (IMS), and 75% of them have not been updated (304).

The hit ratio of the cache is not simulated, but set by the user. The advantage is that the cache doesn't have to be filled up at the beginning. The measurements of the simulation can start immediately. The cache-hit ratio can be found in different experiments and Polygraph offers a hit ratio of 58%. The hit ratio for the memory versus the disks is more complicated. There hasn't been much research done in this area and Polygraph doesn't differ between memory hits and disk hits. One could calculate how many objects are available in cache by divide available memory space with average object size. But this is not always true, since some caching algorithms don't store large files in memory, but in disk. The average object size for a file in memory will then be

less than 11kB. The frequency of the requests for these objects is next problem. If one can store 10,000 objects in memory, what is the probability that next request will be one of those. If the caching algorithm is good, the 10,000 most popular objects will eventually be stored in memory, but again it depends on the caching algorithm. To make this less abstract, a simplified formula for memory hit ratio will be used. If one divides 80% of the available memory space and divides it by the total memory space, a fairly reasonable number comes out. It has been assumed that about 300MB of the memory is used for Linux (60MB), object index (1MB/GB), application (120MB), and temporary variables. The memory hit ratio for various memory sizes can be found in Table 7.



(1)

Table 7: Memory hit ratio for different memory sizes

Memory size	Available memory space	Memory hit ratio
500 MB	200 MB	0.32
750 MB	450 MB	0.48
1000 MB	700 MB	0.56
1500 MB	1200 MB	0.64
2000 MB	1700 MB	0.68

4.2 The client model

The main purpose of the client is to open connections and send requests to a server. It is not supposed to simulate client behavior, but to generate realistic network traffic. The client has a mailbox to handle network communication. When a packet has been created, it is placed in the inbox of the network and the client waits for a response from the server. The network places the response in the inbox of the client. When the whole request has been processed and all packets have been received, either a new request is sent immediately, or the connection is closed. In the latter case a new connection will be opened after a random time. In HTTP/1.0 a client has to open one connection for each request. With HTTP/1.1 it is possibility to pipeline several requests over

one connection. In the model, a client opens one connection and sends a burst of requests. The minimum number of requests is 1 and the user can set the maximum. The final number follows a triangular distribution, where the user can specify the mode. When every object has been received, the client can send a new random number of requests to the same server, or close the connection. If objects from another server are requested, the client has to close the connection and open a new one. A burst of requests is always to the same server. It can never be to different servers in the model. This is done to reduce the complexity. In many cases a web page contains only object from one server, but occasionally it has objects from others as well, e.g., advertising. Since each client can be seen as a connection, another client (connection) can simulate fetching those objects.

The client has a process structure, which keeps track of the transaction. It has an array of request, since there can be multiple request on one connection. The structure can be found in the *declarations* node of the submodel:

```
struct requests{
        unsigned int object_size;           /* The size of the requested object */
        unsigned int bytes_received;        /* Total bytes of object received */
        char done;                          /* If the whole object has been received */
        double sent;                        /* The time when the request was sent */
};

struct{
        unsigned short int server;          /* The server it sends the request to */
        char connection;                    /* The status of the connection, */
                                            /* 1 = open, -1 = closed */
        requests r[MAX_REQUESTS];           /* A structure of requests for this client */
} process;
```

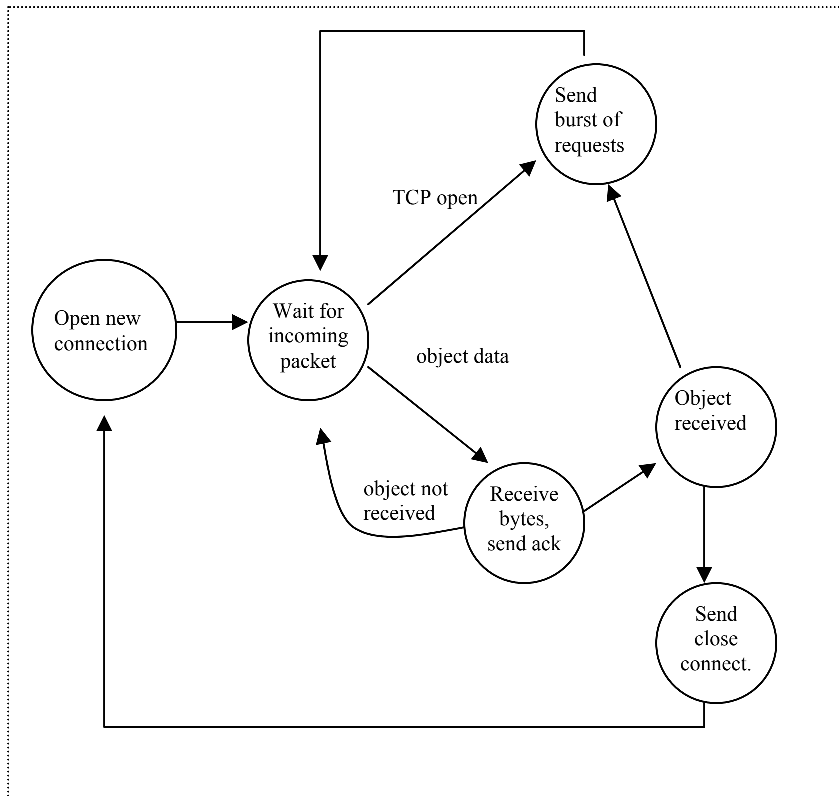


Figure 20: State machine for the client network interface

The state machine for the client is fairly simple. Most of the time the client waits for a packet to arrive. When this happens, the packet has to be processed, and the procedure is decided by the id of the packet (*packet.id*). It has been assumed that the packet processing time for the client is 0, since the network already contains a delay. The only time the client waits is when it decides whether to send a new request or not, and before it opens a new connection.

The implementation in SES/Workbench looks similar to the state machine.

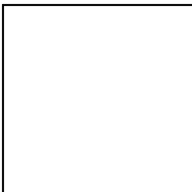


Figure 21: SES/Workbench model of the client

There is only one transaction in the client model and it moves around without any delays. It can only be stopped at three nodes: the *receive_packet* node – where it waits for a packet to arrive, the *think_for_a_while* node – where it decides to either close the connection or send new requests, and the *start_new_request* node - where it waits a random time before it starts a new process. The model looks like a flowchart and corresponding arcs can be found in the state diagram of the client.

The process starts at *start_new_request* and the transaction moves to *set_variables* where a random server is selected and other variables are initialized. At the *TCP_open*, a packet is created and moved to the inbox of the network, *upstream*. It contains the client id, the server id and a packet id. The last one is set to -1, which means TCP-open. The client id is set to the id of this client and the server id to the selected server. The size of the packet is the standard request size (64 bytes), and the info variable is not used at all. The transaction moves on to *receive_packet*, where it waits until the network puts a respond packet in the inbox. The network uses the *packet.client* variable to select to which client it should leave a message, so the client knows that this message belongs to it. When a packet arrives, the *packet.id* is checked, and the transaction can take the following paths:

1. The id is -1, which means TCP-open has been acknowledged. The transaction chooses the *connection_opened* path and sends requests to the server. In SES/Workbench, the same packet is used but *packet.id* will be changed to -2, which means HTTP-request. The *packet.request* is set to 1, and if more than one object is requested, the transaction enters a loop to create additional request packets. The request counter will be incremented to distinguish the requests from each other. The transaction then returns to the *receive_packet* node and waits for the requests to be acknowledged. When a request is sent, the variable *request.sent* is set to the current time, so the total latency can be measured.

2. If a packet has id>0 and the connection is opened, it means the request has been acknowledged and the packet contains the actual data. The transaction moves along the *process_packet* arc and receives the bytes. The *process.r[packet.request-1].bytes_received* is increased by (*packet.size*-58). The packet is acknowledged by sending a 64 bytes packet with id equal to the previous *packet.id*+*packet.bytes*+1, which means, “I haven’t seen id+bytes+1”. E.g., the server sends a packet of 10 bytes starting at 1. The client responds: “I haven’t seen 11”. If all bytes have been received (*process.r[packet.request-1].bytes_received* == *process.r[packet.request-1].object_size*), the client checks if all request have been completed. If so, it either sends a new burst of requests (id = -2) or closes the connection (id = -3). The path is chosen randomly and the user sets the probability. The request service time is collected and sorted according to the hit type. If the request still has bytes to receive (*process.r[packet.request-1].bytes_received* < *process.r[packet.request-1].object_size*), it moves back to the *receive_packet* node and waits for more data.
3. If the packet id is -3, it means that the server has acknowledged the close connection request. The transaction will move along the *close_connection* path and the status of the connection will be set to closed (*process.connection* = -1). The packet then has to be destroyed before the transaction can move to *start_new_request* and wait for a random time before everything starts over again.

4.3 The network model

The network model simply forwards messages from one end to the other with a certain delay. Server/client processing time and packet drop is assumed to be included in the latency, with the exception of server think time - when the server generates a web page. The messages/packets to be transmitted are placed either in the upstream or downstream mailbox, depending on if they are from the client or the server.

The network submodel contains two mailboxes for incoming packets - one for packets from the client, and one for packets from the server. In some cases the cache will be seen as server (client-network-cache) and in some as client (cache-network-server). The network contains two transactions and two paths for full duplex, which means that packets can be sent in both directions simultaneously. The transactions wait at *get_client_message* and *get_server_message* for a packet to arrive. When a packet arrives at *upstream*, the transaction at *get_client_message* grabs the packet and moves to *send_message_upstream*. A new transaction is created and moved back to *get_client_message* to wait for next packet to arrive. The original transaction continues to hold the packet and moves along the network. When it reaches *Instance of network_latency*, is has to wait for a certain amount of time, before it can move on to *leave_message_at_server*, where the message is left either at the cache inbox, or the server [*packet.server*] inbox, depending on which network that has been used. After this the transaction moves to *finish_upstream* and terminates. The transaction at *get_server_message* follows the same flowchart. The network latency has the same distribution and parameters for the network, since it is the same physical connection, just different directions. The network is assumed to work without congestion or loosing any packets. The packet drop rate slows down the incoming packet rate and the network speed decides the delivery time.

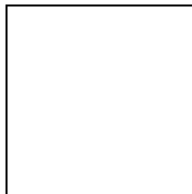


Figure 22: SES/Workbench model of the network

4.4 The server model

The server gets a request, opens a connection and sends the object in form of packets. It can handle several connections at the time, since each connection has a separate process with status

variables linked to it. Each connection uses the client id, since each client can only have one connection. It can also be seen as the connection id. Again, the intention of this submodel is not to simulate server behavior, but to generate realistic network traffic. The process for the server is similar to the process for the client and can be found in the d*eclarations* node of the submodel:

```
unshared struct requests{
        unsigned short int sent_packets;    /* Number of packets sent */
        unsigned int object_size;           /* The size of the requested object */
        unsigned int bytes_sent;            /* Total bytes of object sent */
};

unshared struct{
        short int client;                   /* The client it sends the object to */
        requests r[MAX_REQUESTS];           /* Structure of requests for this server */
} process;

char connection[MAX_CLIENTS];               /* The status of the connections, */
                                            /* 1 = open, -1 = closed */
```

When the server receives a packet, it checks the *packet.id* to find out how to handle it. The server also has to make sure that the right transaction holds the packet. Depending on several variables, the transaction can follow different arcs and a simplified state machine of the server is described in Figure 23. Figure 24 shows the implementation in SES/Workbench.

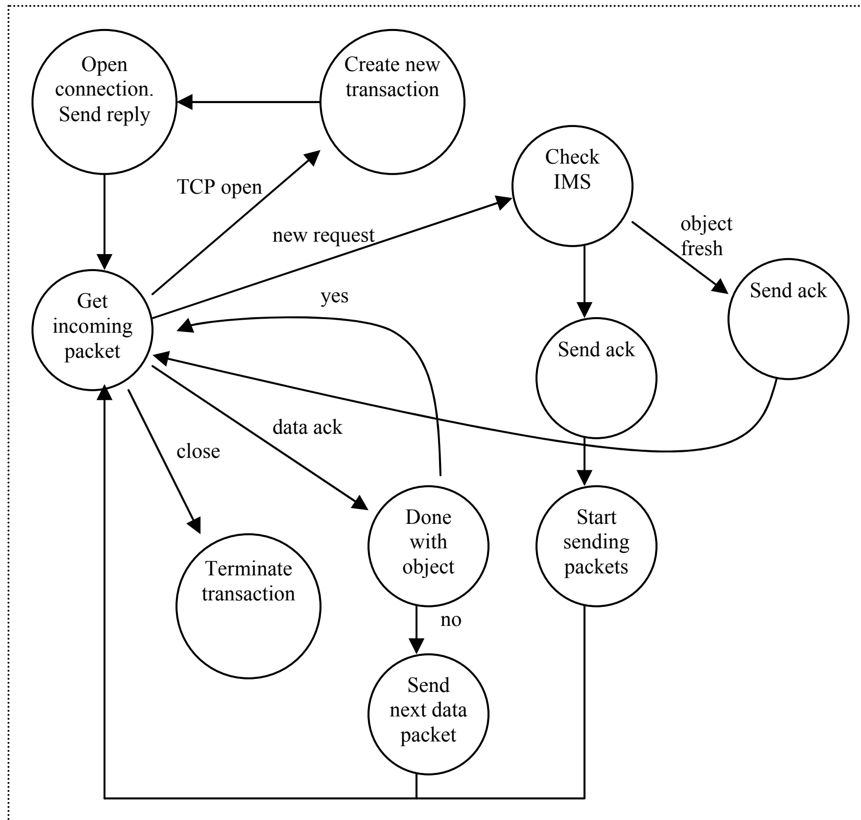


Figure 23: The state machine for the servers

Figure 24: SES/Workbench model of the server

The server model is similar to the client model and it works similar to any client-server application. That means it waits for requests from clients, processes them, and sends a reply. There can be multiple connections and each connection has a transaction and a process tied to it. At the beginning - the idle state - one transaction waits at the *receive_packet* node. When a packet arrives in the inbox, the id is checked and the outgoing arc is chosen depending on its state.

1. If the id is -1 and the transaction grabbing the packet is not handling any requests at the moment, the transaction chooses the *new_connection* path and moves to *process_new_connection.* A new transaction is created, to handle any new incoming

connection and it is moved to the *receive_packet* node. The original transactions goes on to *send_sync* and the packet is sent back to the client with id = -1. The *process.connection* is also set to open, 1. The transaction then moves to *receive_packet*, where it waits for a request from the same client.

2. If the id is -2 and the connection is open, it means that the packet contains a request. If the transaction is the one handling this connection (*packet.client* == *process.client*), it moves along the *get_request* path and creates a thinking request. The request is taken care of by another transaction, since the current one might have to deal with other requests from the same client. The thinking transaction waits for a certain time before it sets the variables of the object. The size of the object is randomly chosen, according to PolyMix-3 distribution. The transaction then tells the server that the thinking has been done by setting the *packet.id* to -4 and returning it to the inbox, before it terminates. If the request is IMS, the processing time is shorter. If the reply is 304, Not Modified, the thinking transaction sends the response directly to the network.
3. The connection handling transaction receives the packet with id -4 and the server starts sending the data. It can send several packets at the time, before it receives acknowledge from the client, and that parameter (*tcp_window_size*) can be changed in the parameter file *constants.txt*. At the node *send_data*, the object is broken down into packets, which needs to be created and parameterized. The *process.sent_bytes* is incremented for each sent packet, before the transaction moves on the *receive_packet* node again.
4. If the transaction that receives a packet is busy with a connection, and the arriving packet belongs to another process, or the receiving transaction is a new transaction but the packet belongs to an already opened connection, the transaction has to release the packet to the waiting transaction. It does that by moving to the *release_packet* node, and puts itself last in the queue at *receive_node*.

5. If the packet id is >0, it means a packet of data has been acknowledged. The transaction moves along the *packet_ACK* path and it can choose two branches. If the whole object has been sent (*process.r[packet.request-1].sent_bytes* == *process.r[packet.request-1].object_size*), it moves to *ack_received*, where the acknowledgement packet is destroyed. If the process has more data to send (*process.r[packet.request-1].sent_bytes* < *process.r[packet.request-1].object_size*), the transaction is moved to *send_packet*, where the a new packet with data is created and the *process.r[packet.request-1].sent_bytes* is incremented. The *packet.id* is set to the first byte of the packet in the total process. E.g. First packet id will be 1, second 1519, third 3037 and so on. The acknowledgement id from the clients will be 1519, 3037 and 4555 respectively ("I haven't seen …").
6. If the id is -3 and the transaction is handling this process, it has to close to connection. It moves along the *close* path to the *close_connection* node, where *process.connection* is set to closed, -1. After that the close acknowledgement has to be sent back to the client, before the transaction can terminates at the *exit* node.

Note that the cache can send a freshness test, or If-Modified-Since (IMS) request. If the object hasn't been modified, the server thinking time is shorter, since it doesn't have to create any dynamic pages or running any scripts.
Since the latency varies for all packets, the server might close a connection before all acknowledgments of an object have been received – if the close connection request arrives before them. In this case, any transaction at the server can remove this packet from the simulation.

The server has an array of connections and they can have two different states:

Table 8 : The states of the process.connection

connection[MAX_CLIENTS]	status
1	open
-1	closed

The value 0 is not used to define any state, since each non-declared variable is set to 0.

4.5 The cache model

The submodel of the cache is the most complex and the most important part. The other submodels focus on generating a realistic traffic flow, to get desired results from the cache. To make the model less complex and easier to understand, it has been broken down into smaller parts. These parts are: *network interface*, *CPU interface*, *disk access, client tasks*, *server tasks*, and *other tasks*.

The network interface handles all incoming packets. They are moved from the network and the mailbox to the network buffer. When the network buffer is full, an interrupt tells the CPU to move these packets to the memory. Figure 25 shows a simplified state diagram of the cache. The two areas for cache hit and cache miss will be described in detail in a later section.

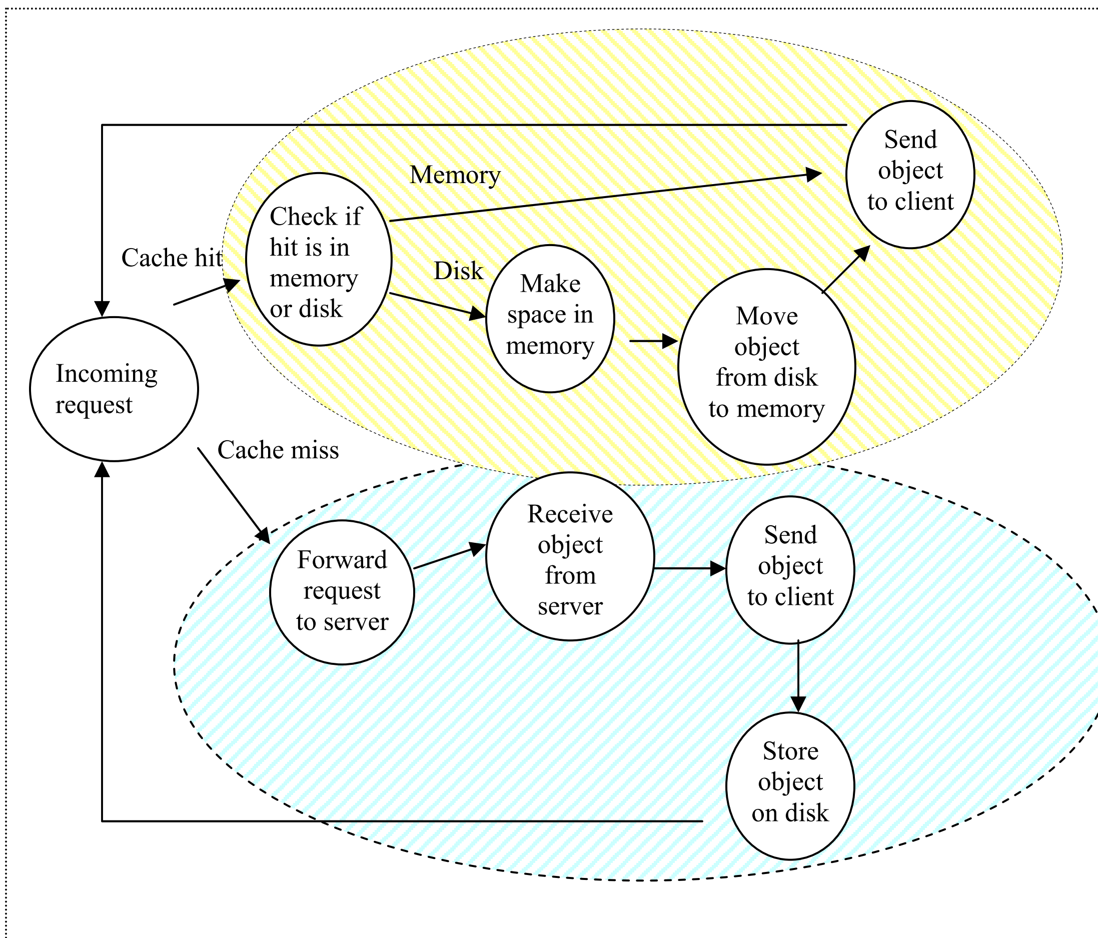


Figure 25: Simplified state diagram for the cache

When the CPU processes a task, it is broken down into 4 different kinds: packet from client, packet from server, NIC to memory transfer, or other transfer. NIC to memory transfer has the highest priority and is taken care of immediately by the CPU. The other processes have their own submodels. The *client_connection* submodel handles all possible actions the CPU has to taken when receiving a packet from a client, while the *server_connection* submodel deals with packets from servers. The *other* submodel handles all other activities the CPU can be imagined to perform. To simulate OS/Application activity, a separate process could create tasks at random times to keep the CPU busy for a certain amount of time. Another approach is to assume that the CPU spends, e.g., 10% of the time for these tasks and have only 90% of CPU speed available for

caching activities. Even though the second approach has been used in the model, the first approach could easily be implemented if desired.

The CPU of the cache has a numerous of variables tied to it. It needs to have information about all current connection statuses, objects being transmitted, and caching status. It has a variable called *kernel*, which tells if the CPU currently is in kernel mode, or user mode. It also carries an array of connections, which holds information about both the connection to the client, and to the server. Since it has been assumed that each client only has one connection to the cache and the burst of request all are to the same server, there is only need for one connection between the cache and the server. If one of the incoming requests is a miss, it will open a connection to the server, and that connection can then be used by future misses from the same client session.

The CPU needs to have one instance of connection structure for each client. Therefore the size of the array is set to *NUM_CLIENTS*. The connection structure also holds and array of requests, since there can be multiple requests on one connection. The requests structure holds information about packets sent, objects sizes, and bytes sent and received.

```
struct requests{
      char sent_packets;              /* Total packets sent */
      int object_size;                /* Object size */
      int bytes_transmitted;          /* Total bytes transmitted */
};

struct connection{
      char client_connection;         /* Status of connection to client */
      char server_connection;         /* Status of connection to server */
      requests r[MAX_REQUESTS];       /* An array of requests */
};

struct{
      char kernel;                    /* Kernel mode, or user mode */
      connections c[NUM_CLIENTS];     /* An array of connections */
} cpu_process;
```

The tasks also have their own data structure. They are similar to the data packet being transmitted over the network, but they have to carry more data. For simplification, a copy of the network packet is included, as well as a set of additional variables with information about the type of task, if the task has been interrupted, where it was interrupted, how long time it had left when it was interrupted, the priority, where it came from, and any other information. All of these variables are

not used the whole time but they have to have a value. SES/Workbench sets all undefined variables to 0.

```
unshared struct{
        packets p;                  /* Holds the network packet if necessary */
        char type;                  /* Type of CPU task */
        char interrupted;           /* If task has been interrupted */
        double_t arrival;           /* Time task arrived at delay node */
        char where[20];             /* Where task is right now */
        double_t service_left;      /* Service time left at delay node */
        char priority;              /* Priority of task */
        char from;                  /* If packet came from CLIENT or SERVER */
        int info;                   /* Additional information about the task */
} cpu_packet;
```

4.5.1 The networking interface

The networking interface simulates the work being done by the NICs. There are two inboxes for incoming packets; one upstream – from the clients, and one downstream – from the servers. To each of these packets a header is added, with information about the origin of the packet – if it's upstream or downstream. The packet carries information about both server and client, so it's not possible to use these fields, since they are the same for upstream and downstream. Each packet is then moved to another mailbox after a certain delay. This delay simulates the frame interval needed when sending Ethernet packets over the network. It depends on the network speed and the frame interval. For Ethernet frames the interval is 20 bytes, and for a 100Mbps network it would correspond to a 1.6 microseconds delay.

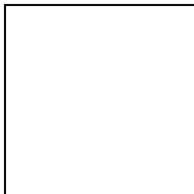


Figure 26: The networking interface

Packets are temporarily stored at the *incoming_packets* mailbox before they are moved to the *NIC_transfer* mailbox, waiting for the CPU to move them to the memory. Between these packet movements, there is a NIC delay, dependent on the NIC speed and the packet size. The NIC

buffer can hold a certain amount of packets and doesn't interrupt the CPU until this buffer is filled, so the CPU can handle several packets at each interrupt. This is called coalescing and the buffer size can be set as one of the parameters. Before the CPU is interrupted, a request of packet transfer is added to the CPU task queue. It has the highest priority (6), which means it will be handled immediately. If the CPU is in kernel mode, it won't be interrupted, but if it is in user mode it will. If the buffer size is large and the traffic is low, a packet may have to wait a long time before it is handled by the CPU. There is a maximum time limit though, since the CPU checks the network buffer continuously, to make sure a connection doesn't get frozen. The frequency of this polling process can also be decided at run time.

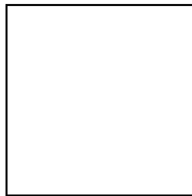


Figure 27: Check the buffer for data

4.5.2 The CPU interface

The CPU interface works almost like a state machine, with many if-then cases, dependent on many variables. The CPU is composed of a transaction, which travels along different arcs, doing numerous tasks with variable delay. The CPU also carries a structure of variables that can be set/changed during the workflow procedure. The first thing the CPU does is to fetch a task. The tasks will be found in a resource pool and they can be: move packet from NIC to memory, process incoming packet, move data from disk to memory, move data from memory to disk, and move packet from memory to NIC. It's also possible to add different kinds of tasks for the CPU. The CPU looks at the *cpu_packet.type* and the *cpu_packet.p.hit_type* variables to find out how to handle the packet. The first variable tells what kind of task it is at the lowest abstraction, and the

second what steps to take at next level. The possible values for the *cpu_packet.type* can be found in Table 9, and for *cpu_packet.p.hit_type* in Table 6.

Table 9: The possible values for the cpu_packet.type

cpu_packet.type	
-2	NIC_TO_MEM
1	MEM_TO_NIC
2	DISK_TO_MEM
3	MEM_TO_DISK
4	SEND_PACKETS
5	SWAP

The tasks can have different priority levels[13], which can be found in Table 10.

Table 10: CPU task priority levels

priority	address	task
1	6	move packet from NIC to memory
2	5	interrupted level 4 or 5 task
3	4	interrupted level 6 task
4	3	move packet from memory to NIC
5	2	disk transfer or other CPU activity
6	1	handle incoming packet

The CPU moves along different arcs depending on the task variables. To simplify the model, the arcs branches out at different locations. The first branch is when the CPU has to decide whether it's a packet from the client, a packet from the server, a request to transmit packets from NIC to memory, or any other task.

---

[13] The priority field in SES/Workbench is used for transactions and not for resources. Since the CPU is only one transaction, using the priority wouldn't help much. A trick is to use the address of the resource pool for priority. When a transaction gets a resource from the pool, it starts looking at the first address (1). If there's nothing there it moves on to next (2). Several resources can be queued in FCFS fashion at each address, and this can be used to simulate task priority. The highest priority will be given the lowest address.

Table 11: First branch for CPU task

Task	Procedure
packet from client	Any incoming packet from client that needs to be interpreted
packet from server	Any incoming packet from server that needs to be interpreted
move packet(s) from NIC to memory	A request to move a certain amount of packets from NIC to memory
other	E.g., memory to disk transfer, or NIC to memory transfer

For each of these tasks, except moving packets from NIC to memory, which is a high priority task, the CPU jumps into another submodel, and doesn't return until it finished or gets interrupted. At that time it either destroys the task, or stores the instant variables before it is returned to the pool (in case the process was interrupted). Any following task, e.g., a reply or disk to memory transfer request should have been made before the CPU transaction left the submodel.

The cache submodel holds several resources, e.g., CPU pool, PCI bus, NICs, and SCSI controller. When a task needs to use one of these resources, the CPU has to allocate it. The task can only continue to be handled if the requested resource is available. Otherwise it has to wait until this occurs. If the cache uses DMA, resources can be held be the SCSI driver and thereby blocking the CPU for some time.

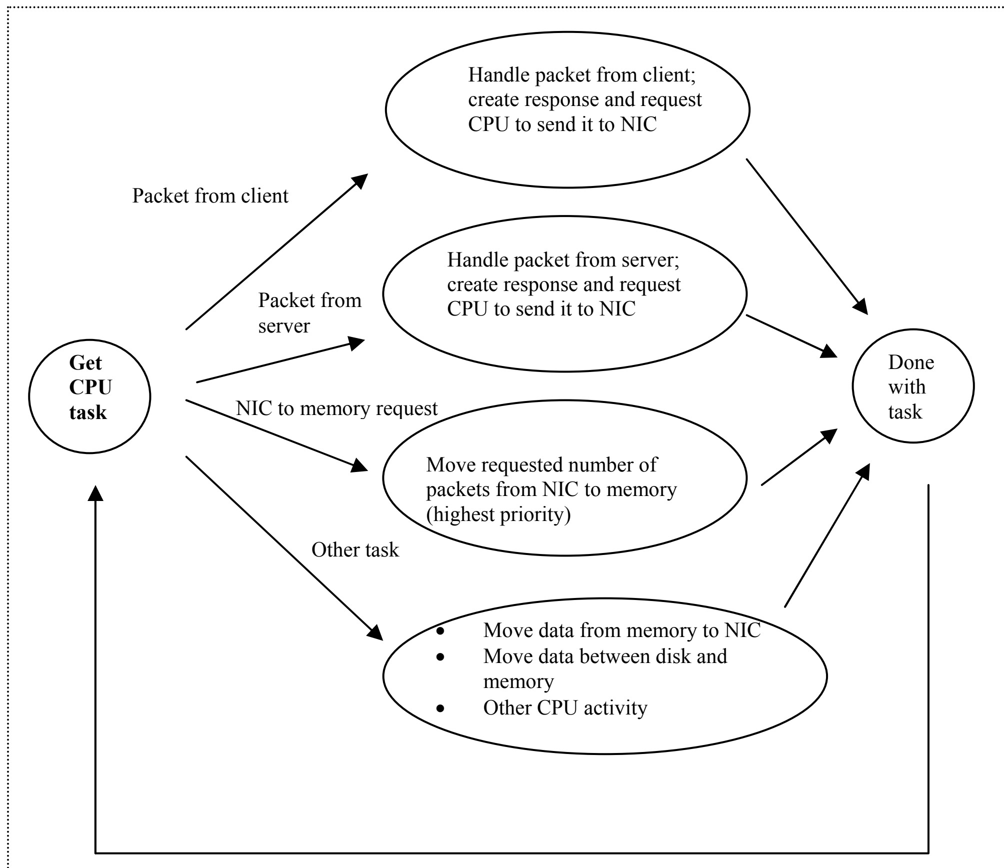


Figure 28: CPU state diagram for cache dataflow activities

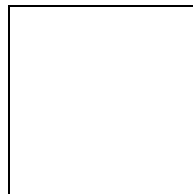


Figure 29: The CPU interface implementation in SES/Workbench

The highest priority task is to move packets from the network buffer to the memory. When enough packets have arrived to the NIC, a request is added to the CPU queue, telling the CPU how many packets that need to be transferred. When the CPU fetches this request (which should be rather soon since it's high priority), it moves along the *NIC_to_mem_transfer* arc and enters the *set_variables* node. At this node the number of packets is registered before the task is destroyed. The CPU then collects one packet and moves it to the memory with a certain delay,

depending on the bus speed, the packet size and the bus width. This procedure goes on until all requested packets are moved. During the transfer the CPU has to allocate both the PCI bus and the memory bus. Since the CPU is in kernel mode during the whole transfer, it can't be interrupted. Requests for transferring more buffered packets can be added to the CPU queue though, but the task will not be dealt with until the current has been completed.

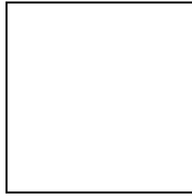


Figure 30: The NIC to memory transfer procedure

4.5.3 The client connection interface

If a packet origins from one of the cache's clients, the CPU enters the *client_connection* submodel for continued processing of the task. The packet id will reveal the nature of the packet. If it's -1 it means open connection, -2 it's a request from the client, >0 the client acknowledges data, and -3 it means close connection. The CPU chooses different arcs depending on this value. To open a new connection, the CPU needs to assign a new port and set up the variables. It also has to create a response packet and put it in the out buffer. It also puts a request in the CPU queue, to send this packet to the network. When this is done, the CPU has completed this task and can return and check the pool for new tasks.

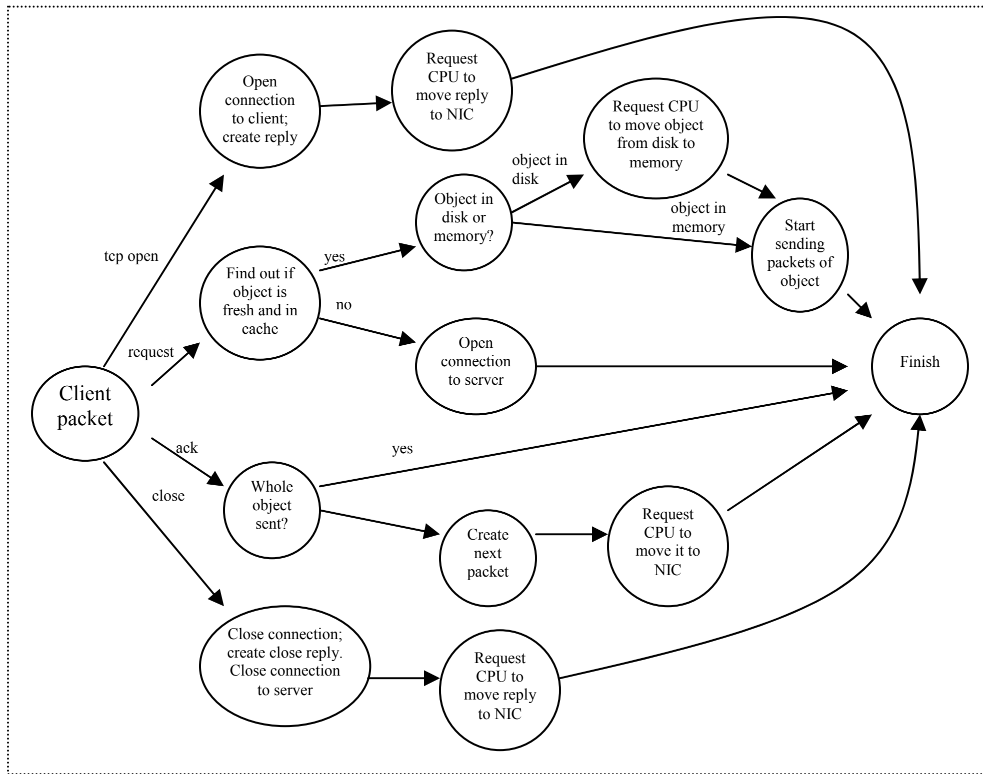


Figure 31: Client connection state diagram

To process a request, much more work has to be done. First the CPU needs to find out if the requested object is stored in the cache or not. A directory of all cached files can be found in the memory. The table also tells the CPU if the object is in the memory or in disk, and expiration date. If it's a miss, the CPU needs to forward the request to the server. That requires several steps and the first one is to send a request to open a new connection to the server. The connection to the server can then be used by succeeding miss requests from the same client.

When the server responds, the following steps will be handle by the *server_connection* submodel, which means that this task is now completed. On the other hand, if the request is a hit, the CPU needs to check that the object is still fresh. If it's not, it will ask the server, which means it has to open a connection to it or use an already open one, if existing. If the object in the cache is valid, the CPU sends a hit reply to the client, with some important information, e.g., the size and the type of the object. If the object is in the memory, the cache can start sending it immediately. If

it's stored in disks, it has to be moved to memory first. The CPU creates a request to itself, to move the object from disk to memory and adds it to the CPU pool. Now the task is completely handled and further actions will be taken care of by the new task.

If the packet id is greater than zero, it means that a sent packet has been acknowledged. If the transfer is complete, nothing else has to be done. The cache can then either wait for a new request or a close connection request. If the whole object hasn't been sent yet, next packet should be created and transmitted. A request to move this data to the network is added to the CPU pool.

When the CPU receives a request to close the connection, the variables have to be reset and an acknowledged packet for the client will be created. It's it added to the CPU pool and will be transmitted as soon as the CPU has time. During all these processes, the CPU can get interrupted – except when it's creating/closing connections. When it has been interrupted, it is moved to the *handle_interrupt* node and all temporary variables will be stored. It means that the CPU can continue with a task where it was and doesn't have to start over at the beginning.

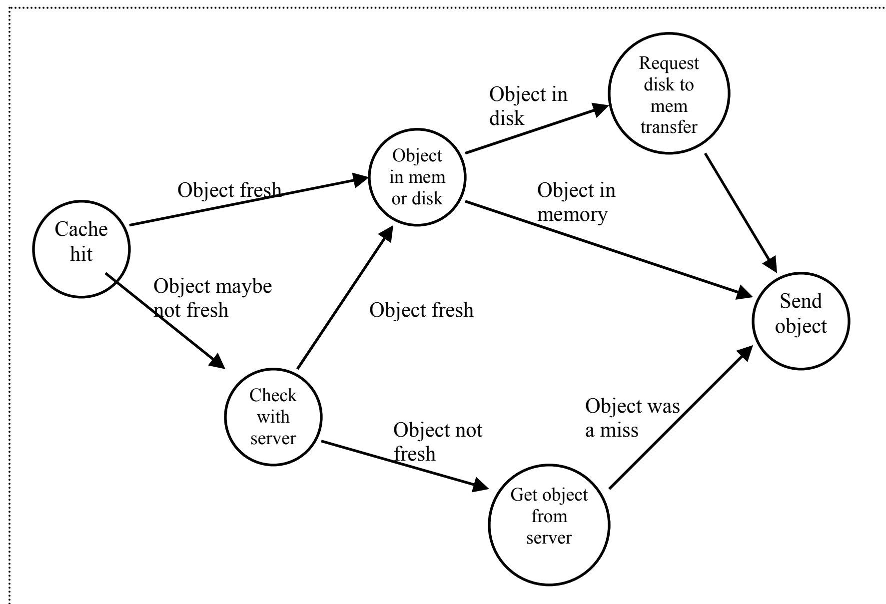


Figure 32: State diagram for cache hit

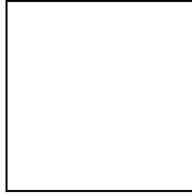


Figure 33: Processing a packet from a client

4.5.4 The Sending packets interface

When an object is in memory it will be broken down into packets and sent to the client. It doesn't matter where the object came from; the server, the disk, or if it already was there. If the object is equal or smaller than 1460 bytes, it can fit in one packet. Otherwise it has to be fragmented into several packets, each with a payload of 1460 bytes. The last packet will probably be smaller. If the object is broken down to several pieces, only some of the packets will be sent at one time. In the TCP settings there is a variable called *tcp_window_size*, which tells how many packets the cache can send, before receiving acknowledgements from the client. When this happens, it can send out a new packet for each acknowledgement.

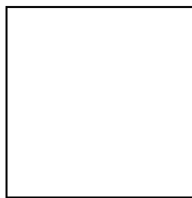


Figure 34: The sending packets submodel

4.5.5 The server interface

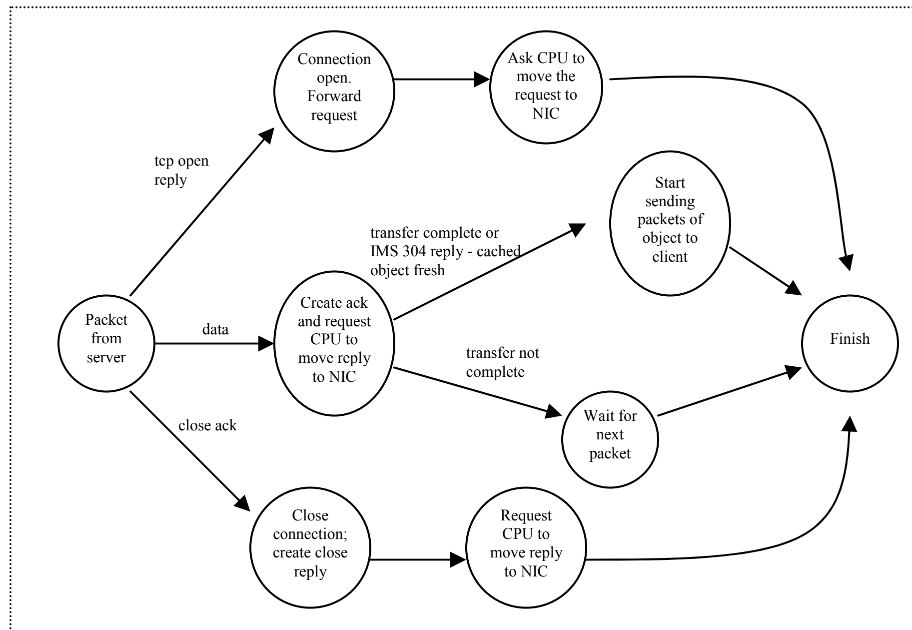


Figure 35: Server connection state diagram

A packet from a server has to be handled in a similar way as a packet from a client. The CPU can either receive a TCP open acknowledgement, which means it should create and forward the request to the server. When the request is acknowledged if first needs to check if it's a reply to an If-Modified-Since request. If it is, and the cache object is fresh (304), it treats the request as a hit. If not, it will be treated as a miss and the cache will start receiving the object data.

When the data packets arrive, they have id greater than zero, and the cache responses with another id, equal to the incoming id plus packet size, plus one, meaning: "I haven't seen this number". The next packer from the server has that id. The packets are stored in the memory and when the whole object has been received, it will be broken down into packets again and sent to the client. When a client closes a connection to the cache, the cache will also close any open connection to the server. When the server responds, the CPU fetches this task and moves along the *get_close_ACK* arc. The connection will be closed and the variables reset. When packets arrive out of order it is handled in software by the TCP layer, which takes some processing time. In the model implementation, all packets will be stored in memory and added to the object

regardless of order. The packet acknowledgement will also be sent to the server. When the object is reassembled in memory the packets can be ordered again. When the cache has received an entire object it can start sending it to the client.

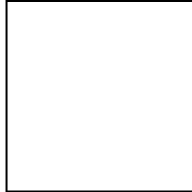


Figure 36: The server connection submodel

4.5.6 The *other* interface

The *other* submodel handles all other tasks that don't involve incoming packets. In the default model that includes memory to NIC transfer, disk transfer, and cache hit handling. Other task that can be included would be, e.g., OS time and application time, but in this model they are assumed to take a constant percentage of the CPU time, which means they slow down each packet handling with a certain percentage. In the long run it gives the same result, but it may vary for each transaction. The percentage can be set as one of the parameters.

If the task needs to access disk, it has to enter another submodel called *disk_access*.

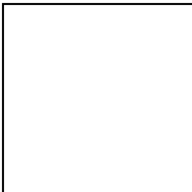


Figure 37: The submodel for handling other types of tasks

4.5.7 The Disk access interface

When the web cache accesses the disk, it can be done in two ways. Either the CPU controls the whole transfer process, or DMA (Direct Memory Access) is being used. The latter case means that the CPU tells the SCSI controller to move the data and can then move on to next task. If any

resource is being used for a data transfer, the CPU has to wait until the resource gets available. If the CPU handles the transfer, it will be tied up during the whole process, and can only move on to other tasks if it gets interrupted. Since the disk access is one of the slowest processes, it is desirable to have the CPU deal as little as possible with it. When the SCSI controller has moved the data it notifies the CPU.

Some times the CPU needs to swap out objects from the memory to find space for new data. The swapped out objects will be stored to disk. The CPU needs to decide which objects to swap, and the decision is made according to the adopted caching algorithm. Examples of popular replacement algorithms are Random, Least Frequently Used (LFU), and Least Recently Used (LRU). The algorithm processing requires substantial CPU time. In the model implementation the user sets a mean value for the number of instructions that need to be executed. The user also decides the percentage of the swap needs.

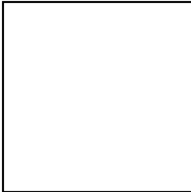
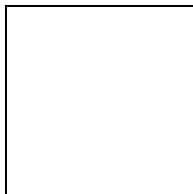


Figure 38: The submodel for disk access

Disk access delay is significant. The average disk time depends on average seek time, average rotational delay, transfer time, and controller overhead.



(2)

Each resource has to be allocated before the data can be moved over the bus. The usage time of the busses are collected and the utilization is printed in the output file.

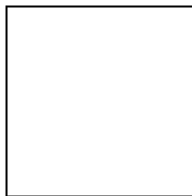
4.6 Delay variables

There are many delay nodes in the model, simulating processing time, buffer fill-up time, network latency, and packet accumulation time. The CPU tasks can only be interrupted at a delay node or a service node so by adding more delay node, more flexibility is given.

The delays can't be set directly. They can only be altered by changing the different parameters. Some of these are network specific and some are configuration specific. There are three different types of delays: CPU thinking time, data transfer, and other delays. The first one depends on the speed of the CPU, the second on other hardware configurations, such as bus width and bus speed, and the third one on user settings.

4.6.1 CPU Thinking time

When the CPU handles packets, it takes some time to find out what type of task it is and to process it. The 16 delay nodes for CPU thinking time can be brought down to 5 basic procedures: processing a task to find out what to do with it, create a CPU request, create a network packet, find out if request is hit or miss, and decide what objects to swap out to disk. The Linux networking stack has been investigated and approximate instruction count is 250, 500, 500, 100,000, and 5,000,000 respectively. It is very hard to come up with exact numbers, since it depends on a numerous of parameters. The instruction count for each delay can be found in Table 12. Basically the delay time in millisecond is calculated by dividing the number of instruction by the available CPU speed. The Cycles-Per-Instruction (CPI) is assumed to be 1.



(3)

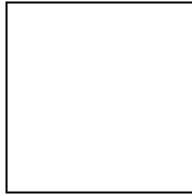
Table 12: Delays for CPU thinking time

Delay name	Submodel	Procedure	Instructions
process_task	cache	disassembly packet, find out if it's from cache, server, NIC to memory transfer request, or other request	250
process_client_task	client_connection	find out if what kind of packet from client it is	250
open_connection	client_connection	open tcp connection, create a reply packet and a request for CPU to move the packet to NIC	750
get_request	client_connection	find out if object is hit (in cache) or miss, and create either reply packet or tcp open to server and a request for CPU to move it to NIC	101,000
get_ACK	client_connection	receive object bytes, create next packet to send to client and a request for CPU to send it to NIC	1000
get_close	client_connection	close connection, create a reply packet and ask CPU to send it to NIC	1,250
close_server	client_connection	create a close packet for server and a request for CPU to send it to NIC	1,000
process_server_task	server_connection	find out if what kind of packet from server it is	250
open_connection	server_connection	open tcp connection, create a reply packet and a request for CPU to move the packet to NIC	1,250
get_reply	server_connection	get object parameters	250
create_ACK	server_connection	receive object bytes, create ack packet and a request for CPU to send it to NIC	1,250
create_reply	server_connection	create a reply packet and ask CPU to send it to NIC	1,000
create_request	server_connection	create a CPU request to send an object from memory to disk	500
create_cpu_request	other	create a request for CPU to move data from disk to memory	500
caching_algorithm	other	find out where to make space in memory and allocate it	5,000,000
swap	disk_access	create a request for CPU to move data from memory to disk	500
reply	send_packets	create a network packet and a request for CPU to send it to NIC	1,000

4.6.2 Data transfer

The data transfer delays depend on the size of the data being transferred, the bus width, the bus speed, and the availability of the resources. Most of the traffic will be between memory and NIC, but it will be showed that the disk access is most significant, since it takes the longest time. There is no delay for transferring data between memory and CPU, since the behavior of this is very unpredictable. The CPU might flush registers and caches for other activities, which means the data has to be transferred again. The research for this project doesn’t go that deep into CPU

behavior so the memory-CPU transfer delay is included in the CPU thinking time, i.e., the number of instructions for each task. There is also some buffering going on at certain locations in the cache box, but it will also be neglected, since it has been assumed that it doesn't slow down the data movement significantly. Bus transfer delays are calculated by taken number of bits divided by available bus width multiplied by bus speed.



(4)

Delay name	Submodel	Procedure
interpacket_delay	cache	receive a packet on the network and give it to the NIC
NIC_delay	cache	move the incoming packet to the network buffer
check_buffer	cache	polling
NIC_to_PCI	cache	move data from NIC to NB over PCI bus
PCI_to_mem	cache	move data from NB to memory over memory bus
mem_to_NIC_transfer	other	move data from memory to NB over memory bus
move_to_NIC	other	move data from NB to NIC over PCI bus
mem_to_PCI	disk_access	move data from memory to NB over memory bus
move_to_SCSI_bus	disk_access	move data from NB to SCSI controller over PCI bus
write_to_disk	disk_access	write data to disk
read_from_disk	disk_access	read data from disk
move_over_PCI	disk_access	move data from SCSI controller to NB over PCI bus
write_to_mem	disk_access	move data from NB to memory over memory bus

Table 13: Data transfer delays

4.6.3 Other delays

The other delays involve client and server thinking time, and network latency. They will be set directly by the user. The client thinking times are exponential distributed, and the user sets the mean value. The network latencies for the client-cache connection and the cache-server connection are normal distributed, and the user sets both mean value and standard deviation. The server thinking time depends on if it's If-Modified-Since request, or regular request. Both thinking times are normal distributed and the user sets both mean value and standard deviation.

Table 14: Other delays

Delay name	Submodel	Procedure
think_for_a_while	client	decide to either send new burst of requests or close connection
start_new_request	client	open new connection
network_latency	network	send packet over network
server_thinking	server	find object, process request, create packets

4.7 Model Parameters

The parameters that can be altered to vary the cache architecture are one of the most important parts of the model. They can be changed and several simulations can be compared to each other to find an optimized architecture. All hardware architecture dependent parameters are stored in a text file named *parameters.txt* and the model reads the values from that file before it starts the simulation. The file also includes explanations and ranges of the parameters.

The parameters decide the duration of all processing done inside the cache. The workflow has to pass through several delay nodes, all depending on the cache hardware. Therefore it is also important that the network works fine and creates a reasonable stimulus. It also has many adjustable variables. They control the traffic flow over the network and parameters in the client and server, e.g., the network latency, the server think time and the client request rate. The variables can be set according to existing network conditions before the simulations are conducted. During the simulations, the hardware parameters can be altered to test the effect of different architectures.

There is not a variable for request rate; it all depends on how many clients there are, and their thinking time. The variables will be found with explanations in the file *environment.txt*

Table 15: Adjustable parameters of the model

Variable	Default	Explanation
CACHE CONFIGURATION		
other_CPU_activity_percentage	0.10	Percentage of available time CPU spends for OS, application or other activity
cache_hit_ratio	0.46	Total hit ratio for cache
memory_size	1000	Memory size in MB
number_of_disks	8	Number of hard disks
size_of_each_disk	8000	Size of each hard disk in MB
CPU_speed	933	Speed of CPU in MHz
mem_bus_speed	133	Speed of memory bus in MHz
mem_bus_width	64	Width of memory bus in bits
NIC_speed	100	Speed of NIC in Mbps
number_of_NICs	2	Number of NICs
NIC_PCI_bus_speed	33	Speed in MHz of PCI bus between NIC and bridge
NIC_PCI_bus_width	64	Width in bits of PCI bus between NIC and bridge
SCSI_PCI_bus_speed	33	Speed in MHz of PCI bus between SCSI and bridge
SCSI_PCI_bus_width	32	Width in bits of PCI bus between SCSI and bridge
NIC_op_width	32	NIC operational width in bits
SCSI_op_width	32	SCSI operational width in bits
SCSI_speed	160	Speed of SCSI bus in MB/s
average_disk_seek_time	4.0	Average time to seek disk in ms
disk_rotation_speed	10000	Disk rotation speed in RPM
disk_controller_overhead	0.5	Disk controller overhead in ms
disk_read_speed	160	Speed of hard disk reads in MB/s
disk_write_speed	160	Speed of hard disk writes in MB/s
DMA	1	If DMA, 1-YES, 0-NO
swap_percentage	0.20	Percentage of swaps from memory to disk

Table 16: Traffic flow parameters

Variable	Default	Explanation
TRAFFIC FLOW		
number_of_clients	600	Number of total clients (connections)
number_of_servers	300	Number of total servers
mean_new_connection_delay	5000	Mean time of how long time it takes for client to decide to open a new connection in ms
mean_new_request_delay	2500	Mean time of how long time it takes for client to decide whether to close connection or send new request (ms)
client_close_connection	0.30	The probability that a client closes a connection, instead of sending a new burst of requests
max_requests_connection	8	Maximum number of requests on one connection
mean_request_connection	1	Mean number of requests on one connection
lan_bandwidth	100	Bandwidth between cache and clients in Mbps
mean_lan_latency	1	Mean latency on network between client and cache in ms
std_lan_latency	0.5	Standard deviation for network latency between client and cache
network_latency	10	Bandwidth between cache and servers in Mbps
mean_network_latency	40	Mean latency on network between cache and server in ms
std_network_latency	20	Standard deviation for network latency between cache and server
packet_loss_percentage	0.005	Percentage of packet loss on the network
tcp_window_size	3	Number of packets that can be sent out before receiving acknowledgement
NIC_buffer_space	8	Number of packets to fill up a buffer, before interrupting CPU
check_buffer_interval	100	How often the CPU should check the NIC buffer in ms
server_think_time_mean	2500	Mean server processing request time in ms
server_think_time_std	1000	Standard deviation for server processing request time in ms
check_freshness_percentage	0.2	How often the cache needs to check for freshness
freshness_percentage	0.75	How many of these IMS that are fresh
server_freshness_time_mean	1000	Server mean thinking time for fresh objects
server_freshness_time_std	250	Standard deviation for that thinking time
i_process_task	250	#instructions to find out the type of task
i_create_CPU_request	500	#instructions to create a CPU request
i_create_network_packet	500	#instructions to create a network packet
i_hit_or_miss_average	100,000	#instructions to find out if object is hit or miss
i_replacement_policy_mean	5,000,000	#instructions to find out what objects to swap out

Table 17: Object parameters

Variable	Default	Explanation
OBJECTS		
image_percentage	0.65	Percentage of image files
html_percentage	0.15	Percentage of HTML files
download_percentage	0.005	Percentage of download files
other_percentage	0.195	Percentage of other files
image_cachability	0.80	Cachability of image files
html_cachability	0.90	Cachability of HTML files
download_cachability	0.95	Cachability of download files
other_cachability	0.72	Cachability of other files
image_size_mean	4500	Mean size of images in bytes
html_size_mean	8500	Mean size of HTML files in bytes
download_size_mean	300000	Mean size of download files in bytes
download_size_std	300000	Standard deviation for size of download files
other_size_mean	25000	Mean size of other files in bytes
other_size_std	10000	Standard deviation for size of other files

Table 18: Other environment parameters

Variable	Default	Explanation
OTHER		
printouts	1	1:YES, 0:NO
kernel_writes	1	If kernel writes/reads from disk, 1-YES, 0-NO
caching	1	Caching type, 1-Delayed store, 2-Immediate store

5 Results

5.1 Validation

The model has been tested extensively during the development, and to verify the model several simulations have been conducted. The results have been compared to simulation results from Polygraph with similar settings. There are many parameters to change for different simulations and corresponding changes on hardware should give matching results. When a simulation is finished, a file called *"cachebox.rpt"* is created and the following variables can be investigated:

Cache resource utilization

- CPU utilization
- NIC-PCI bus utilization
- SCSI-PCI bus utilization
- Disk utilization
- NIC utilization
- Network utilization

Cache system performance results

- Requests per second
- Total request response time
- Request response time for memory hits
- Request response time for disk hits
- Request response time for misses

The CPU utilization is the total time the CPU is holding a task and handling it, divided by the total simulation time. It includes cache thinking time and all bus transfer time, unless DMA is

being used. In the latter case, the CPU won't be tied to disk-memory transfer, but it might be stalled for a short time, in case any needed buses are being held for the transfer.
Since the newer architecture has two PCI buses – one for storage disks and one for network controllers, there will be two different variables to look at. If the current hardware only has one shared PCI bus, the model could be changed without too much complication to implement the desired architecture.

The NIC and the network utilization are also important factors, since they tell how much load that can be put on the cache. Even if one tries to add more traffic, it can only be received at the speed the network allows. If the utilizations go up too high it will mean that the maximum absorbable traffic is being induced. The network utilization is not how much traffic that flows over the network, since it was assumed that the network would never be congested. Instead the usage time is the interframe distance, divided by network speed. It tells the delay between two packets and sets a limit for how fast packets can arrive to the NIC.

The requests-per-second is an important factor of the model. It is one of the parameters in Polygraph, and the simulation tool creates virtual clients and servers to induce the desired network traffic. In the SES/Workbench model, the user has to specify the number of clients and the number of servers. The request rate can then be alternated by changing the client think time – i.e., how long time it takes for a client to decide to send new requests. A client doesn't start thinking until all previous requests have been completed, which means that the requests-per-second rate is dependable – not only on the number of clients/server and the think time – but also on the cache performance. If the latency is high, it will take longer time for a certain client to send out new requests, and the request rate will drop. This means that the user will specify the number of clients and their thinking time and then find out how many requests per second the cache can handle.

The response times tell how long time it takes for a request to be handled, i.e., how long time it takes for an object to be sent to the client. The client action prediction is complicated and

in this model the clients just add network traffic. What happens if a request takes too long and a client gets tired of waiting is out of the scope of this thesis. When the request rate increases it will take longer time before the cache gets to a certain request, since it has to handle many packets at the time. It will also make the latencies increase, so the relation between request rate and latency has to be investigates carefully. The latencies can be compared with Polygraph simulations and they should give matching values. By observing these variable the user can find out where the bottleneck of the web cache is. If the CPU utilization is high it might mean that a faster CPU could increase the performance. It can also mean the any of the buses is slow, since the CPU is tied at data transfer, so these variable have to be investigate too.

A few simulations were conducted and the results were compared with similar simulations with Polygraph to validate the model. The parameters for the SES/Workbench model and Polygraph were set to match each other. The resulting request rate for the model was given as a parameter for Polygraph, and the resulting hit ratio for Polygraph was given as an input to the SES/Workbench simulation. CPU speed and the hardware configuration were the same. Using the results, the parameters for the model can be validated and justified.

PCI bus timings were captured and statistics were generated using Hewlett-Packard E2927A PCI Bus Analyzer. SysStat under Linux was used to capture system resource utilizations, such as, CPU, and disk. Unfortunately, SCSI Analyzer was not available to measure details on the SCSI bus timings.

It wasn’t possible to compare the PCI bus utilization for the SES/Workbench model and the Polygraph simulations, since the simulating web cache had a shared PCI bus for NICs and storage disks. Both resources had to compete of the bus, which significantly reduced performance. An assumption for the model was that the CPU was busy with other tasks such as OS and maintaining 10% of the time. That leaves 90% of the CPU power for Web caching. The

measurements on the hardware include both maintenance and caching activity. The hardware for the validating simulation is the default settings found in Table 15.

Table 19: Results from the validating simulation

	SES/Workbench model	**Polygraph/hardware measurements**
CPU utilization	11.3%	9-11%
Memory bus utilization	0.6%	X
NIC-PCI bus utilization	3.5%	10.4%
SCSI-PCI bus utilization	0.9%	7.8%
Disk utilization	89.5%	X
NIC utilization	34.8%	X
LAN utilization	0.3%	X
Network utilization	1.4%	X
Requests per second	203.0	199.9
Total average request response time	1649.6ms	1469.6ms
Request response time for memory hits	176.5ms	X
Request response time for disk hits	262.9ms	X
Average hit response time	215.6ms	X
Request response time for memory hits (No IMS)	8.9ms	X
Request response time for disk hits (No IMS)	60.5ms	X
Average hit response time (No IMS)	31.6ms	34.1ms
Request response time for misses	2784.2ms	2644.7ms
Cache-hit ratio	46%	46.31%

Polygraph doesn't differ between memory hits and disk hits since it doesn't communicate with the cache box. It just tells the total hit response time for the cache. It does distinguish the If-Modified-Since request though, both for 200 (Modified) and 304 (Not modified). The 304s are not included in the total hit time for Polygraph, but it is for the SES/Workbench model. To validate the hit response time, the simulation had to be conducted without IMS requests. The memory hit response time and disk hit response time were 8.9ms and 60.5ms respectively. The average would then be 31.6ms. If the IMS 304 (Not-Modified) responses are included, the hit response times will be 176.5ms for memory hits and 262.9ms for disk hits.

The Polygraph hardware has a shared PCI bus for NIC and SCSI, which will be more utilized than the two PCI busses in the model. The CPU utilization for the model does not include the OS and maintenance overhead. It has been assumed that the CPU spends about 10% of the

time handling other activities. All hardware measurements have been made during Polygraph simulations and an approximate value for the utilization is achieved by dividing the used time by the total time for this measurement. It is noticeable that the hardware didn't have any problems handling the caching procedures for 200 requests per second.

5.2 Additional Experimentation

The following SES/Workbench simulations show that improving disk access speed is more important than improving CPU speed. For the test, the speed of the CPU was increased to 2GHz and a new simulation was conducted. The results can be found in Table 20.

The faster CPU doesn't improve cache performance at all, since the disk access is the bottleneck. The CPU is less utilized, which means it can process more packets. The SCSI controller gets more work to do and reduces the caching performance. The average request response time actually increases slightly due to slow disk response time. The memory hits and the misses are almost unaffected by this change. The disk hits don't take longer time to handle, but the queue for accessing the disk will be congested and thereby slow down the disk response time. A problem here is that the disk reads don't have priority over disk writes. Every miss means a write to the disk when the new object arrives, which will compete with the disk hits over the SCSI controller. This does not affect the memory hits and misses.

Table 20: Results from the faster CPU simulation

	Original simulation	**Faster CPU**	**Faster disks**
CPU utilization	11.3%	7.1%	12.0%
Memory bus utilization	0.6%	0.6%	0.5%
NIC-PCI bus utilization	3.5%	3.7%	3.4%
SCSI-PCI bus utilization	0.9%	1.0%	0.9%
Disk utilization	89.5%	93.7%	72.3%
NIC utilization	34.8%	36.8%	34.1%
LAN utilization	0.3%	0.3%	0.3%
Network utilization	1.4%	1.5%	1.4%
Requests per second	203.0	201.8	210.1
Total average request response time	1649.6ms	1652.5ms	1600.4ms
Request response time for memory hits	176.5ms	173.5ms	171.7ms
Request response time for disk hits	262.9ms	334.9ms	200.5ms
Request response time for misses	2784.2ms	2758.9ms	2752.0ms
Cache-hit ratio	46%	46%	46%

A simulation with faster disks (average seek time 2ms) instead of faster CPU was conducted and the results show that the performance improves significantly. The average response time has been reduced, mostly due to faster disk response time. It made the request rate increase to over 210 requests per second. This indicates that it's more important to have faster disk access than a faster CPU for this architecture. Another way to try to reduce response time could be to change the caching algorithm, to try to reduce the number of disk writes and reads.

To investigate how response time varies with request rate, a number of shorter simulations were conducted. Due to limited hardware resources, only simulations with less than 300 requests per second could be run with Polygraph. The results indicate that the average response time increases rapidly, but not as fast as for Polygraph. Since Polygraph doesn't differ between memory hits and disk hits, it's not possible to find out if both the memory hits and the disk hits increase in time. The miss response time is almost constant with the SES/Workbench model. The hardware model is using a shared PCI bus for NIC and SCSI. Therefore network packets can't be moved to memory when the SCSI is using the PCI bus. Slow disk access will slow down disk hits - and network traffic, which increases response time for, not only disk hit, but also memory hits, and

misses. Figure 42 confirms this theory. It shows that both the miss response time and the memory-hit time for the model are close to constant and that only the disk hit time increases. It's using two different PCI busses, so the network traffic won't have to wait for the SCSI to release the PCI bus.

In practice, higher request rate will mean longer response time for both misses and memory hits. More TCP connections mean longer connection setup time and more memory used. The memory-hit ratio should therefore decrease when less memory is available for storing objects. When the response time is calculated in the model, the TCP connection setup time is not included. The timer starts when a request is being sent over an already open connection and stops when the entire object has been received.

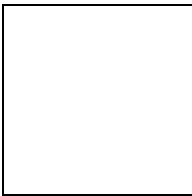


Figure 39: Average response time comparison

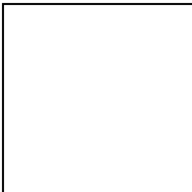


Figure 40: Hit response time comparison

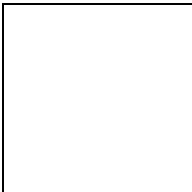


Figure 41: Miss response time comparison

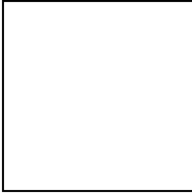


Figure 42: Request response time for the SES/Workbench model

6 Conclusion

6.1 Summary

A novel method, modeling a Web cache server system in SES/Workbench environment, was developed to gain better understanding of a web cache server internal workflow and system bottlenecks. The model uses statistical parameters and transactional state machines to capture the timing characteristics of a web cache system hardware and software. Similar method can be used to analyze trade-off's in a new system architecture, to evaluate performance of existing system configurations, and to evaluate caching software algorithms.

The model provides a small but adequate set of parameters that can be altered to tune the model, so that a desired system behavior that matches a physical system's can be recreated. This method is useful when details of the caching software (storage, replacement, freshness) algorithms are unknown. The first challenge is in defining this set of parameters to accurately represent the system behavior. The second challenge is in acquiring these parameters from a physical system for validation purposes, since present systems do not provide adequate facility to probe both in hardware and in software. For example, the network workload model is similar to the traffic generated by Web Polygraph clients and servers. However, the network workload did not accurately represent details of the object mix and did not account for processor service times in clients and servers. Consequently, the model results indicate slightly higher response time than the hardware measurements. It is a daunting task to capture enough microlevel information, such as PCI bus timings in nano- and microseconds details, to generate statistical parameters that will be used in hours of simulation environment, where results are measured for milliseconds and seconds values.

The main difference between the model and Polygraph is that Polygraph is using object id and fills up the storage with these objects. The SES/Workbench model is built upon probabilities,

which means there is no need for fill up time. The measurements can start almost immediately. The number of servers is not important, since it has been assumed that the server can handle unlimited requests at the same time, without additional delay. That means that the servers will never be overloaded.

For the disk access, different caches implement different methods. In the model, one can choose between delayed store and immediate store. The latter case stores an object from server to disk as soon as it has been sent to the client. It can be really hard to find out what algorithm a certain machine uses, and for the hardware comparisons it seemed like the cache didn't store objects immediately, but waited some time and wrote a burst of objects at the same time.

The clients of the model try to send requests with some delay and can't sent further request until the previous ones have been handled. For Polygraph, a certain request rate is set, and clients and servers are justified to keep that rate constant. With well-set parameters the SES/Workbench model shows similar results as the Polygraph simulations, and the workflow inside the box corresponds to actual measurements. The few differences can depend on caching algorithms, data overflow, and coincidences. The conclusion is that the model is fairly close to Polygraph network traffic and server traffic flow hardware measurements. By supporting the ability to vary both hardware configuration and network traffic, the simulation model is a valuable tool for testing the workflow in a web cache server.

SES/Workbench generates statistics every time the simulation is run, but not completed requests will not be included. Usually that will be the long jobs, which may give them lower average response time. To minimize the effect of this, one can increase the simulation time.

The results show that the model catches the trends of the workflow in a cache box. Even if the bus utilization for the model doesn't correspond completely with the hardware results, it shows the relationship between the traffic and how changes affect different areas. It also makes it possible to isolate the bottleneck and study it separately. Since the utilization of the busses was hard to measure, it was simplified by dividing busy time by total time for a measurement. It's

important to remember that the results from the hardware measurements also include all additional traffic associated with OS and other activities.

6.2 Model extensions

The model uses generated traffic from clients and server to look at the workflow within the web cache server. It is specified on the hardware inside the box and doesn't focus much on the actual traffic and algorithms. To extend the model further, there are many things that can be done. At the current time, the cache-hit ratio and the memory-hit ratio are given as parameters. The truth is that they actually depend on the memory size, the disk size, and the type of traffic. Some parts of the memory are being used for OS, Web caching application, and a list of stored objects, and some is used for caching some of the objects. The bigger the memory is, the more objects can be stored, which means the memory-hit ratio will increase. The same is for storage disks. The more space available, the more objects can be stored, which will increase the cache-hit ratio. To simulate this in SES/Workbench, each object could be given an id number (from 1 to total number of web objects available) and the cache could keep a list of all objects cached and if the object is in disk or memory. It will extend the usage of the models, since different caching algorithms can then be tested. The workflow between memory and disk will be more realistic and the cache will handle real objects, instead of being dependent on probabilities. The biggest problem is the memory usage of the simulation model. If the user wants the request rate to be 250 requests per second and every request takes about 1.5 seconds, it means that the cache has to deal with 375 requests at each time. If each requested object is 11kB (average object size), it will make about 3000 packets (375*8), not counting TCP overhead and the packets for internal work inside the cache. Each client and server holds at least one transaction and a data structure. The transactions are large data structures requiring approximately 4kB of memory each [11]. The additional data structure, *process*, also requires much memory. This should be multiplied with the total number of clients and servers. If one wants to add id number, expiration date, and a variable

for where an object is stored, the memory will increase further. The cache will be empty at start and needs some time to fill up memory and disks before the actual measurements can start. Polygraph needs 4 hours to do this and then the simulations run for about 16 more hours. With the workflow model, where the probabilities are set, the simulation can be run for a much shorter time, since there's no need for filling up memory. This will reduce simulation time significantly.

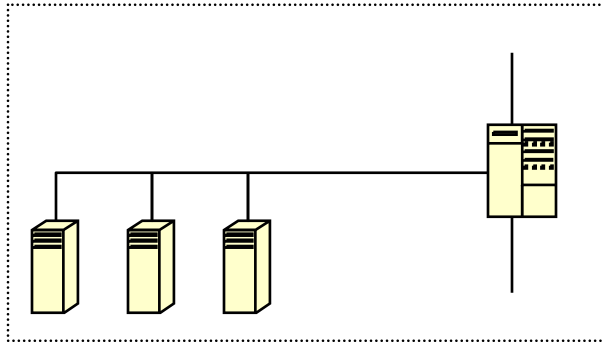


Figure 43: Load balancing

It is also possible to add more groups of clients to either the existing network or to a new one, connected to the cache. Another group of servers could be added in the same way. It is also possible to cluster several caches for either peer caching or load balancing. The only thing that has to be added is a device to control the traffic by setting up certain algorithms.

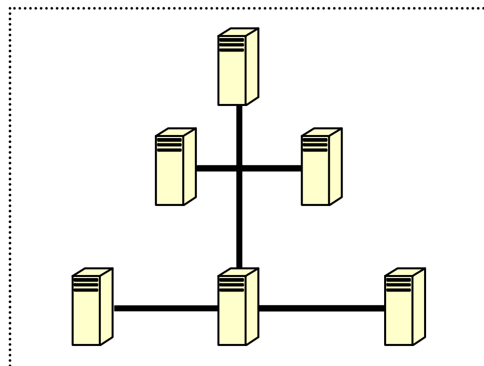


Figure 44: Cache clustering

The parameters for CPU think-time are based upon Linux networking stack. Each function has been broken down into hardware language and an approximate value of the number of instructions has been calculated. In practice it's really hard to find out how long delays the CPU

think-time really are, since it also depends on the caching algorithm as well as the hardware. Further investigations could be made to get better approximations of these instruction counts. This goes as well for other parameters, such as cache-hit ratio, memory-hit ratio, freshness-test ratio, and freshness ratio. For this simple model these values have been set according to previous research.

The model could also be modified for other similar areas, such as Network Attached Storage (NAS). Web caching and NAS are closely related, so the changes needed in the model are not too complicated. The clients could still be used to generate traffic – by loading and storing data on the cache. Some data might be in memory, some in disk, and some at another location. In this case the clustering and load balancing become even more interesting, since several servers can be connected in any way the user wants.

References

[1] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. *Web caching and Zipf-like distributions: Evidence and implications*. IEEE Infocom, vol. XX, No. Y, Month 1999. http://citeseer.nj.nec.com/breslau98web.html

[2] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., 1973.

[3] Gary Tomlinson, Drew Major, and Ron Lee. *High-Capacity Internet Middleware: Internet Caching System Architectural Overview*. Sigmetrics 3-2000, 27:4.

[4] Alec Wolma, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. *Organization-Based Analysis of Web-Object Sharing and Caching*. The 2nd USENIX Symposium on Internet Technologies & Systems. Boulder, Colorado, USA, October 11-14, 1999.

[5] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley Professional Computing Series, 1994. Page 30.

[6] Alex Rousskov, Duane Wessels. *The Third Cache-Off: The Official Report*. October 11, 2000.http://www.measurement-factory.com/results/public/cacheoff/N03/report.by-meas.html

[7] William Stallings. *Data and Computer Communications, Fifth Edition.* Prentice Hall, 1997.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. June 1999. http://www.w3.org/Protocols/HTTP/1.1

[9] T. Berners-Lee, R. Fielding and H. Frystyk. *Hypertext Transfer Protocol -- HTTP/1.0*. RFC 1945, May 1996.

[10] J. Reynolds, and J. Postel. *Assigned Numbers*. STD 2, RFC 1700, October 1994.

[11] HyPerformix, Inc, *SES/Workbench 3.3 Quick Start and Upgrade Guide*,

[12] David Baker, Russell Haddleton, and Kevin Wika. *A Distributed Scheduling Simulation*. IEEE Infocom, 3/92.

[13] R. Hariharan, W.K. Ehrlich, D. Cura, P.K. Reeser. *End to End Performance Modeling of Web Server Architecture*. Sigmetrics 9-2000, 28:2.

[14] Mitchell L. Loeb, Andrew J. Rindos, William G. Holland, and Steven P. Woolet. *Gigabit Ethernet PCI Adapter Performance.* IEEE Network, March/April 2001.

[15] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian Bershad. *Workloads for Programmable Network Interfaces*. Electronic Services.

[16] Toby J. Teorey, David W. Bachmann, Thomas A. Makowski, Mandyam M, Srinivasan. *User Profile and Workload Analysis for Local Area Networks*. CITI-TR-90-3. June 15, 1990.

[17] Martin F. Arlitt and Carey L. Williamson, *Web Server Workload Characterization: The Search for Invariants.* 1996 ACM Sigmetrics Conference, Philadelphia, PA, May 1996.

[18] Carlos Maltzahn, Kathy J. Richardson, Dirk Grunwald. *Performance Issues of Enterprise Level Web Proxies.* Sigmetrics '97 Seattle, WA, USA.

[19] Ron Lee and Gary Tomlinson. *Workload Requirements for a Very High-Capacity Proxy Cache Design.* Workshop papers, 1999.

[20] Anja Feldman, *Web Performance Characterization*, AT&T Labs-Research. Available from http://www.research.att.com/~anja/

[21] Arthur Goldberg, Ilya Pevzner, and Robert Buff, *Caching Characteristics of Internet and Intranet Web Proxy Traces*. Available from http://www.cs.nyu.edu/artg/research/proxymodel/proxymodel.doc

[22] James Rubarth-Lay, *Optimizing Web Performance*. Available from http://www.ddg.com/LIS/CyberHornsS96/j.rubarth-lay/PAPER.html

[23] Jean-Chrysostome Bolot and Philipp Hoschka, *Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design.* Available from http://www5conf.inria.fr/fich_html/P44/Overview.html

[24] Bruce A. Mah, *An Empirical Model of HTTP Network Traffic.* Available from http://citeseer.nj.nec.com/mah97empirical.html

[25] Allison Woodruff, Paul M. Aoki, Eric Brewer, Paul Gauthier, and Lawrence A. Rowe. *An investigation of documents from the World Wide Web,* In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May 1996.

[26] Mark E. Crovella and Azer Bestavros, *Explaining World Wide Web Traffic Self-Similarity.* Available from http://www.cs.bu.edu/faculty/crovella/paper-archive/self-sim/paper.html

[27] George Kingsley Zipf, *Human Behavior and the Principle of Least Effort.* Hafner Publishing Company, New York, NY, 1949

[28] Duke P. Hong, Celio Albuquerque, Carlos Oliveira, and Tatsuya Suda. *Evaluating the Impact of Emerging Streaming Media Applications on TCP/IP Performance.* IEEE Communications Magazine, April 2001.

[29] Jussara Almeida and Pei Cao, *Measuring Proxy Performance with the Wisconsin Proxy Benchmark*, J. of Computer Networks and ISDN Systems, 1999.

[30]. Paul Barford and Mark Crovella, *Generating representative web workloads for network and server performance evaluation.* In Proceedings of the Joint International Conference on Measurements and Modeling of Computer Systems (Sigmetrics ‘98/Performance ’98), pages 151-160, Madison, WI, June 1998.

[31] M. Baentsch, A. Launer, L. Baum, G. Molter, S. Rothkugel, P. Strum, *World-Wide Web caching: The Application-Level View of the Internet,* IEEE Communications, June 1997.

[32] George Bilchev, Chris Roadknight, Ian Marshall, and Sverrir Olafsson, *WWW Cache Modelling Toolbox.* Workshop papers 1999.

[33] Web Polygraph, http://www.web-polygraph.org/

[34] Reza Rejaie, Mark Handley, Haobo Yu, and Deborah Estrin, *Proxy Caching Mechanism for Multimedia Playback Streams in the Internet.* Workshop papers 1999.

Appenix A: SES/Workbench

Modern dynamic systems - including information systems, computer networks, hardware and software, and business processes - behave in complex ways that are difficult to understand. This difficulty typically leads to many errors and inadequacies in developing such systems and planning for their use. Since the cost of delaying the correction of these errors grows rapidly with time, the value of early problem detection and correction is high. Simulation provides a practical way to detect such problems and allow early correction. Avoiding the use of simulation substantially increases the risk of failure. Simulation is a proactive approach of solving problems with complex dynamic systems, which evolve in complex ways over time. Potential problems with these systems involve performance, cost, correctness, completeness, consistency, reliability, security, time-to-market requirements, usability, and maintainability. Approaches to solving problems with complex dynamic systems include simulation, measurement, benchmarking, rules of thumb, analytic modeling, and reactive problem solving.

SES/workbench is a simulation-modeling tool for hardware architecture and other complex systems. Used by the major electronic, telecommunications and computer companies in the US, Japan and Europe, it is an industrial strength modeling and simulation package that enables one to model large, complex systems and get timely, useful results that impact the design and business.

A.1 SES/Workbench collection

SES/Workbench is an integrated collection of software tools for specifying and evaluating system designs. Workbench consists primarily of three components:

- SES/*design*™ - a graphical editor module for specifying a systems design.
- SES/*sim*™ - a translation and simulation module for converting the design specification into an executable simulation model. The language is based on C and C++ and does not

require the use of SES/*design*, although it is highly recommended, since it tremendously simplifies the design process.

- The Animated Simulator (SES/*scope*™) – which provides the ability to observe and debug an executing simulation model.

A.2 SES/Workbench benefits

- *Reduced risk and uncertainty* - accurately builds an executable model of current systems as well as future configurations
- *Optimized cost and performance* - model can be built with quantitative comparisons between the current process and proposed processes to determine the most efficient and cost-effective alternative
- *Validated system design* - animated business processes confirms functionality and workflow
- *Reduced time-to-market* - identifies problems early in the development cycle
- *Managing evolving systems* - provides clear understanding of the next step

A.3 System Design

The Graphical Interface allows users to build and represent system designs pictorially. It also includes:

- Management of active, passive and logical resources
- Transaction flow control, including concurrency and synchronization
- A critique facility for early detection of errors before compilation
- A browser for searching and replacing

A.4 The Animated Simulator

Among the most important features of SES/*workbench* is the ability to animate a model so that customers can observe the tasks as they flow through the system. Features include:

- Real time graphing capabilities
- Real time inspection of statistics, variables, transactions and nodes
- The ability to set breakpoints
- A programmable tracing mechanism
- Soft buttons to control the speed and views of the model
- The ability to save the environment you have define

A.5 Model Components

SES/Workbench is a transaction-based simulation tool, based on queuing. The **transactions** are the basic entities representing execution of processes in a Workbench model. A transaction flows from node to node, manipulating resources and executing statements. At each node, the transaction gets processed (executes some code), departs the node along an arc, flows across the arc to the next node, and either begins its processing or enters a queue for that node (if other transactions are already queuing). It is important to understand that the execution flow is event-driven, with the events being scheduled before the simulation starts. All random numbers and distributions have the same seed if a simulation is run again, which makes it possible to regenerate the same results. When a transaction passes along an arc, it traverses with no delay and code can be executed so the transaction can change state or phase.

In general, a node in a Workbench graph can represent any sort of processing the modeler wishes. However, there are four general groups of nodes that should satisfy all modeling needs:

- Resource-management nodes
- Transaction flow-control nodes
- Submodel-management nodes
- Miscellaneous nodes

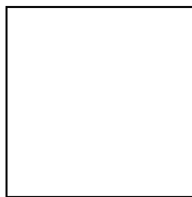


Figure 45: Submodel page nodes

A.5.1 Resource-management nodes

There are four kinds of resources: *Active*, *Passive*, *Data*, and *Logical* Resources. The active resources are, conceptually, something that actively processes instructions or data (e.g., a processor or disk controller). An active resource is represented by a Service node in Workbench. A passive resource doesn't do any work itself, but it's something that must be possessed by a transaction to do work. Example of passive resources are memories, buffers, and database locks. Passive resources usually occur in groups called "resource pools". For example, memory can be regarded as a pool of pages. The individual members of each resource pool are called "resource pool elements". The data resource is an addressable passive resource, each element of which carries an unshared datum. Data resources particularly are useful in modeling messages in message-passing systems, with the unshared datum representing the message. Data resources can also be used to reduce the number of transactions flowing through a model. A logical resource is

an arbitrary condition, upon which a transaction has to wait, e.g., a Boolean expression. The transaction waits until the condition is true and then departs the node. The Block node provides this logical resource mechanism.

There are 12 different types of resource-management nodes:

Block node

The Block node provides a mechanism for making a transaction wait upon an arbitrary condition. A transaction arriving at a Block node enters the queue and waits until the specified Boolean condition is satisfied. If the condition is satisfied and a server is available, the transaction departs. If the condition is not satisfied, the transaction waits (is blocked) until the condition is reevaluated.

Service node

The Service node represents a device designated to perform a specific function for many users. The node is used to model an active resource – often a hardware device such as a disk drive or CPU.

Allocate node

The Allocate node is use to allocate resources to transactions (often in conjunction with the Release node).

Create node

The Create node is used to create new elements of a resource pool. When a transaction arrives at a Create node, the specific number of new elements is added to a specific pool.

Resource node

The Resource node is used to specify one of several types of passive resources, depending upon the specific needs of the modeler. Each passive resource pool is represented by a Resource node.

Release node

Transactions arriving at a Release node releases some or all of the resources they hold, either to the original resource pool, to another pool or to another transaction. The resources can also be destroyed.

Destroy node

The Destroy node is used to remove free elements from a resource pool. When a transaction arrives at a Destroy node, the number of elements specified by the Destroy node is removed from a specific resource pool.

Set node

The Set node provides a mechanism for changing the power of Allocate, Block, and Service nodes or to change the priority of selected transaction residing in those nodes.

Delay node

A Delay node is used to delay a transaction for a specified amount of time. When a transaction arrives at a Delay node, it delays for the time specified, without queuing, and proceeds. It can be used to simulate propagation delay in hardware or when unlimited active resources are available.

Finite Queue Service node

The Finite Queue Service node implements a custom method to set a limit on the maximum number of transactions that can be at the node. If a transaction arrives at a Finite Queue Service when the queue is full, its symbolic port name is set to *Overflow*, and the transaction bypasses the

node without receiving service. A separate topology arc can be connected to the exit port of the node so that transactions with a port variable of *Overflow* are rerouted after bypassing the node.

Finite Queue Allocate node

The Finite Queue Allocate node implements a custom method to set a limit on the maximum number of transactions that can be at the node. If a transaction arrives at a Finite Queue Allocate when the queue is full, its symbolic port name is set to *Overflow*, and the transaction bypasses the node without receiving service. A separate topology arc can be connected to the exit port of the node so that transactions with a port variable of *Overflow* are rerouted after bypassing the node.

Finite Queue Block node

The Finite Queue Block node implements a custom method to set a limit on the maximum number of transactions that can be at the node. If a transaction arrives at a Finite Queue Block when the queue is full, its symbolic port name is set to *Overflow*, and the transaction bypasses the node without receiving service. A separate topology arc can be connected to the exit port of the node so that transactions with a port variable of *Overflow* are rerouted after bypassing the node.

A.5.2 Transaction flow-control nodes

Transaction flow-control nodes are used to control the model transaction population, and to direct the flow of transactions through the system. The nodes used to control the transaction population do so by creating and destroying transactions. These nodes are: *Source* node, *Fork* node, *Split* node, *Sink* node, and Join node. There are also 4 transaction flow-control nodes that alter transaction flow trough the system (or to define a point on a graph, in the case of the Branch node): *Branch* node, *Loop* node, *Interrupt* node, and *Resume* node.

Source node

The Source node provides a means of injecting transactions into a model after initialization. (Fork and Split node also allow post-initialization injection.)

Sink node

All transactions arriving at a Sink node simply disappears from the model. Any resources held by the transactions are returned to their owning resource pools.

Fork node

Fork node provides an additional means of injecting transactions into a model after initialization and a means of synchronizing parallel processes. When a parent transaction arrives at a Fork node, it creates a number of children transactions. The parent and children transactions then leave the Fork node one at a time. The Fork node works together with the Join node.

Join node

A Join node represents a point in a model where a parent transaction waits for all its children transactions to rejoin with it, before it can continue. Actually “rejoin” is a somewhat misleading term, since the children disappear at any Join node they enter, regardless of whether their parent is waiting at that or *any* Join node. Any resources held by the children when they disappear are returned to the resource pools owning those resource elements.

Split node

The Split node, like the Fork node, is used to inject new transactions into a model after initialization. Unlike the Fork node, however, the Split node provides no synchronization mechanism.

Loop node

The Loop node, as its name suggests, is used to route transactions repeatedly through a particular section of a model. Transactions loop through that section until some termination condition is satisfied. Transactions can be looped under for, while, or do{}while conditions.

Interrupt node

The Interrupt node is used to interrupt the processing of one or more transactions at other nodes. It's also possible to change the state vector of the interrupted transactions. When a transaction arrives at an Interrupt node, the transactions identified by the node's specification immediately stop their processing and leave the nodes at which they reside. The transaction causing the interrupt also immediately leaves the Interrupt node.

Resume node

The Resume node is used to route interrupted transactions to a specific point in a model. If a Resume node is placed in a submodel, any transaction interrupted in that submodel are placed immediately at the Resume node, where they execute the node's method and immediately depart.

Branch node

The Branch node is used simply to define a point in a graph, usually from or to which arcs are led; it performs no processing on transactions. It's mostly used to make graphs neater and more easily interpreted, or to collect desired statistics about the specific points in the graph that they define.

A.5.3 Submodel-management nodes

Submodel-management nodes provide the mechanism by which models can be constructed as a hierarchical collection of submodels. They are used to represent submodels and to move transactions from one submodel to another. There are 4 different submodel-management nodes:

Enter node

When a transaction arrives at a Submodel Reference node, it is immediately moved to the Enter node of the submodel referenced. The Enter node represents simply the point of entry into a referenced submodel.

Call node

The Call node is similar to the Submodel Reference node in that it allows movement of transactions from one submodel to another. However, it differs, since it can contain multiple calls embedded in the C control-flow statements, it allows named node instances to be called in the same submodel as the Call node, and it's not limited to calling submodels.

Submodel node

Depending on where it is placed on a Module page or on a Submodel page, and on how it is specified, a Submodel node can be placed as *Definition* node, *Type* node, *Instance* node, or *Reference* node. They are similar but differ in how the relate to a submodel they are connected to.

Return node

When a transaction arrives at a Return node, it is moved immediately to the Submodel Reference node or Call node, from which it originally moved into the submodel. The Return node represents simply the point of return to a calling submodel or Call node.

A.5.4 Miscellaneous nodes

The Miscellaneous nodes are nodes that don't fit other categorizations. They include the *User* node, *Declaration* node, and *Super* node.

User node

The User node allows use of C and the SES/*sim* language to specify arbitrary computations to be executed. Users include the necessary declarations and statements using the Method Options form. When transactions arrive at the User node, they execute the statements specified in the node method.

Declaration node

The Declaration node is used to declare miscellaneous constants, variables, and routines using C. These are usually independent nodes (no arcs required).

Super node

The User node can perform the characteristic statements of any node or combination of nodes, except the Service node (and there are certain restrictions on the use of **allocate**, **allocate_until**, and **block_until** statements). The intended use of the node is for those situations where transactions should have data added to them or some actions performed before they leave the node and enter the model and the resulting combination of nodes is cumbersome.

The User node and the Super node are similar in that neither has a characteristic statement of its own and both are used for C or SES/*sim* language entry. They differ in the fact that a User node prohibits entry of any characteristic statement from other nodes, whereas the Super node permits any (except Service).

Appenix B: The Zipf distribution

Zipf[14] curves follow a straight line when plotted on a double-logarithmic diagram. In the figures, the same data is plotted on linear and logarithmic scales. Both plots show a Zipf distribution with 300 datapoints.

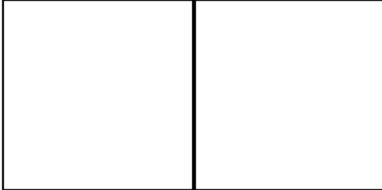


Figure 46: Zipf distribution with linear scales and logarithmic scales on both axes

The formula for the Zipf distribution (Zipf's law) is:

$$f(w) * rank(w) = constant \quad (3)$$

where the rank increases along the x-axis and the frequency along the y-axis. That means that the second ranked object is half of the highest ranked.

A simple description of data that follow a Zipf distribution is that they have:

- a few elements that score *very* high (the left tail in the diagrams)
- a medium number of elements with middle-of-the-road scores (the middle part of the diagram)
- a huge number of elements that score very low (the right tail in the diagram)

[14] The Zipf distribution was published by George Kingsley Zipf 1949

Much available data suggests that web use follows a Zipf distribution. The figure shows the distribution of incoming page requests to www.sun.com during a one-month period. Each data point represents one page, with the x-axis showing pages sorted according to popularity: the first page is the most popular one (the home page), the second page is the one that received second-most requests that month, and so on until page number 10,000 is reached, which was only requested a single time that month. The heavy line shows the actual empirical data from the log files and the thin red line shows a Zipf curve that seems to fit the data quite well except for the low end. The deviation at the low end is due to a variety of factors, including the fact that the site is not old enough yet to have enough accumulated pages of low-frequency interest.

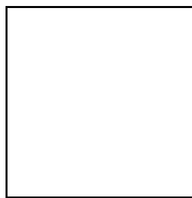


Figure 47: Page popularity

The figure shows ***incoming*** page-requests to a single site. Other studies have found that Zipf curves also characterize the *outgoing* page requests from the employees of an organization (there are a few pages that everybody look at and a large number of pages that are seen only once). It also seems that the distribution of hypertext references on the web follows a Zipf distribution (both in the sense that there are a few sites that everybody link to and many sites that almost nobody links to; and that any given site gets much of its traffic referrals from a few other sites while receiving small amounts of traffic from a vast variety of other sites) and that the participation in Usenet discussion groups follows a Zipf distribution (a few people post most of the messages and many people post very sparingly).