Differentiated Services in Switching-based Web Caching System

by

JIAN ZHOU

A thesis submitted to the School of Computing

in conformity with the requirements

for the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

June 2003

0-612-81141-7

Abstract

Differentiated services (DiffServ) are being adopted in a wide number of Internet applications, including web services. In the web-caching field, researchers have proposed to achieve DiffServ on original web servers, cache servers, or at the client side. We argue that there are significant advantages of achieving DiffServ on another kind of nodes - edge routers - in a web caching system. Edge routers can perform request classification, and assign the type of service, hence the per-hop Behavior (PHB) of the classified requests. If it maintains content information for each cache server, then the edge router is able to provide quality of service to different requests by forwarding the requests to the most appropriate cache server from the clients' point of view, while balancing the workload of cache servers.

We propose a Switching-based Differentiated Service Caching (SDSC), the goal of which is to provide three different types of service to three classes of requests, namely Streaming-object Class, Real-time Assured Class and Best-effort Class. A detailed simulation model is described. We show that for streaming object requests, if cache servers have enough disk-bandwidth and cache size, then most SC requests meet the DiffServ requirements in terms of response time. We also show that the Acceptance Rate (AR) of SC is not influenced by the request intensity of RC and BC. Compared with BC, the average response times of RC are lower than that of BC. Furthermore, the workload of all cache servers is well balanced.

Acknowledgements

I would like to express my gratitude to my supervisors Dr. Hossam Hassanein and Dr. Patrick Martin, for their enlightening guidance and instruction without which this thesis is impossible.

I would like to thank all members of the Telecommunications Research Lab and the Database Lab at Queen's University for their suggestions, assistance, comments and friendship.

Special thanks to my wife Yaping and my son Carl, who constantly supported, encouraged me to do my best.

Finally, I would also like to extend my appreciation to the School of Computing and Queen's University for providing such great study environment and all-aspect support. My experience at Queen's University is unforgettable.

List of Acronyms

AR	Acceptance rate of SC requests
BC	Best-effort Class
CGI	Common Gateway Interface
DB	Delay Bound
DBR	Disk-bandwidth Routing
DSCP	Differentiated Service Codepoint
SDSC	Switching-based Differentiated Service Caching
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ICP	Internet Cache Protocol
IETF	The Internet Engineering Task Force
LB_L5	Load Balancing Layer 5 switching-based transparent
LRU	Least Recently Used algorithm
NLANR	National Laboratory for Applied Network Research
MRT	Minimum Response Time
PHB	Per-hop Behavior
QoS	Quality of Service
RC	Real-time Assured Class
RTT	Round Trip Time
SC	Streaming-object Class
SLA	Service Level Agreement
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
WR	Workload Routing
W3C	World Wide Web Consortium

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

The increasing diversity of Internet applications calls for differentiated services for different types of applications. For instance, IP voice and video services require high-quality support in terms of low delay, low jitter and great bandwidth while email and file transfer have low quality requirements. The dramatically increasing number of clients accessing the Internet, combined with limited network resources, also means it is desirable to provide differentiated Internet access, providing clients with a range of service-quality levels at a range of prices. For this purpose, IETF has introduced differentiated services – DiffServ [25], whose structure and scheme has been adopted in a variety of Internet applications, including web applications.

Web proxy caching is the key performance accelerator in the Web infrastructure [13]. It can reduce latency by satisfying a request from a cache instead of the origin server. It can also reduce traffic as each object is only retrieved from the server once, thus reducing the amount of bandwidth used by a client. Finally, it can provide differentiated service (DiffServ) to network clients and applications.

To achieve DiffServ in web caching, many research efforts have addressed performance differentiation on web servers or cache servers. On web servers, resources (CPU, memory, disk or network bandwidth) can be allocated according to DiffServ requirements of different clients. For instance, Abdelzaher, Shin and Bhatti [1] presented an approach to resource management based on web content adaptation.

Service differentiation can also be achieved on cache servers. Several policies have been devised for optimally allocating disk storage to static files, such as bias replacement in favor of important users. For example, Lu, Saxena and Abdelzaher [31] proposed a resource management architecture for web proxy caches that allows differentiating hit ratio values seen by different content classes.

In an enterprise domain, DiffServ can be achieved on edge routers. Since an edge router possesses the ability to categorize incoming requests into different classes, and to assign per-hop behaviors (PHB) to the requests based on classification, an edge router is a good place at which to achieve DiffServ for web caching. LB_L5 [30] is a Load Balancing Layer 5 switching-based transparent Web caching scheme that forms the basic structure for our research, with switches as edge routers. In LB_L5, a switch forwards an incoming request to one of the caches it connects with according to the content and workload of the caches and network latency. LB_L5, however, does not guarantee the minimum response time. The Minimum Response Time (MRT) scheme [45] extends LB_L5 to guarantee the minimum expected response time for requests.

Our research has two main goals. The first goal is to study how DiffServ can be achieved in a switching-based web caching system. The second goal is to study how streaming-object requests can be serviced in this system, and how the DiffServ requirements of such requests can be satisfied.

The motivation of the research is to propose an end-to-end web-based DiffServ system. The system is integrated in that DiffServ is achieved at ISO Application layer, Transport layer and Network layer. In this thesis, we are concerned with achieving DiffServ at Application layer, which utilizes DiffServ at Network and Transport layers. DiffServ at Network layer can be achieved by adopting IETF's DiffServ model [26].

To this end, we propose the Switching-based Differentiated Service Caching (SDSC), which contains two major components: request classification and request routing at the switch. Request classification categorizes client requests into three classes – Streaming-object Class (SC), Real-time Assured-object Class (RC) and Best-effort object Class (BC), based on request content type. Class information is then carried on in the IP DSCP byte [26]. After request classification, a switch uses the IP DSCP information and routing algorithms to forward the request to the most appropriate cache sever, if the DiffServ of the request can be achieved. Otherwise the request is dropped. Request Routing uses three routing algorithms, namely, Disk-Bandwidth Routing (DBR) for streaming media, Minimum Response Time (MRT) for real-time assured class objects, and Workload Routing (WR) for best-effort class objects.

The main contribution of the research is twofold. First, we studied differentiated service in web caching, and proposed a solution (SDSC) to classify HTTP requests and then achieve DiffServ for the classified requests. Second, a comprehensive simulatior was developed to evaluate the performance of the proposed SDSC.

The rest of the thesis is organized as follows. Chapter 2 provides a review of related work. Chapter 3 presents an overview of the proposed SDSC scheme and describes the two major components of the scheme – Request Classification and Request Routing. Performance evaluations of the proposed scheme SDSC are conducted and related models are presented in Chapter 4. Chapter 5 concludes the thesis and suggests future work.

Chapter 2 Related Work

Distributed web caching is one of two most popular types of cooperative cache system [35], with the other one being hierarchical system [35]. This chapter presents a brief review of the two systems, introduces and compares several cache server selection algorithms, two of which are adopted in our scheme. We then introduce several differentiated service implementation schemes proposed by other researchers that realize DiffServ on cache servers or web servers.

Streaming media caching [41] is another major concern of the research. Streaming media is increasingly popular in current web applications, and caching it poses unique problems because of its large size and real-time requirements. We introduce prefix caching, a popular method of caching streaming media, which is adopted in our work.

SDSC extends the LB-L5 structure [30], which in turn is based on the transparent distributed web caching architecture [16]. Transparent caching is a variant of web caching where web traffic is automatically intercepted and redirected toward one or more web cache servers, using L5 switches. The switch transparently intercepts web traffic originated by the clients and applying a load balancing policy redirects http requests toward one of the available cache servers.

2.1 Distributed Web Caching and Cache Selection Algorithms

The two popular types of cache server cooperation are hierarchical and distributed web caching. In hierarchical cache server architecture, each cache server in the hierarchy is shared by a group of clients or by a group of child cache servers. In distributed web caching, however, cache servers are organized into cache clusters with no definite hierarchy among them, and are allowed to be distributed geographically over large distances, as shown in Figure 2.1. A device, such as a switch [6] or a local cache, sits between the client cluster and the cache server cluster in the distributed web caching architecture.

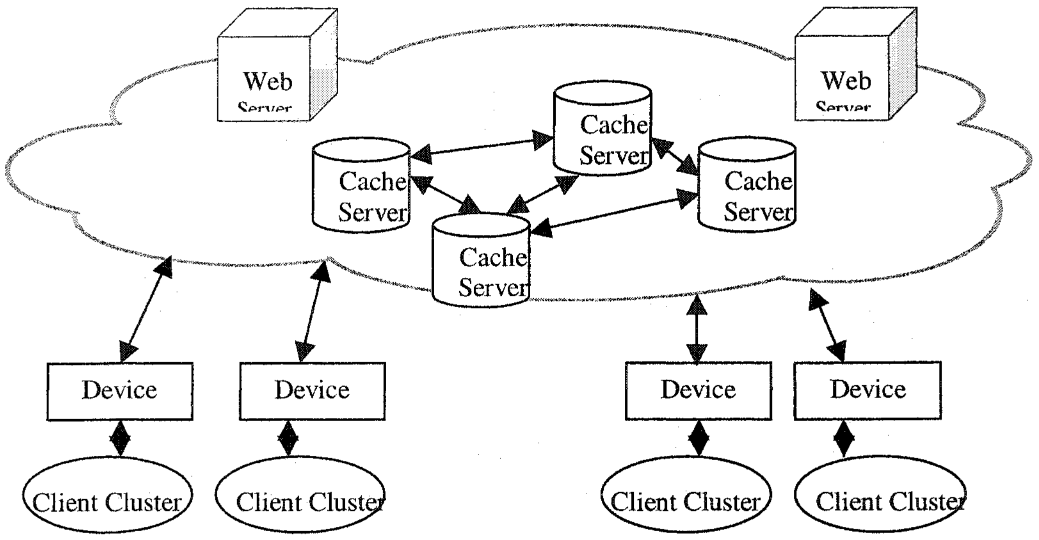


Figure 2.1 Distributed Web Caching Architecture [6]

Distributed web caching handles a data request in the following way: if a local cache server contains the data requested by a client, the server sends the data to the client. Otherwise, the local cache server or a switch device redirects the request to one of the cache servers. If that cache server has a copy of the requested object, it sends the data back to the client; otherwise the request is redirected to the web server. Compared with the hierarchical architecture, distributed web caching systems have several benefits such as distribution of server loads and improvement of client performance by bringing cache servers closer to the client.

Popular cache server selection algorithms fall into two categories: client- initiated selection and switch selection. Among these selection algorithms, Minimum Round Trip Time and Minimum HTTP Request Latency [38] are client- initiated, while Content-based Selection [12], Workload-based Selection [14], LB-L5 Selection [30] and MRT [45] Selection are switch based.

2.1.1 Minimum Round Trip Time

The round trip time (RTT) of a request uses the Ping utility to determine the proximity of distributed servers. The Ping utility uses the Internet Control Message Protocol (ICMP) [27] to send ECHO_REQUEST to the cache server's echo port and listens to the ECHO_RESPONSE. The ping round trip time reflects the actual network load on the route between the client and the server.

The NLANR [35] log traces used for our simulation use RTT to measure the response time of packets. The drawback of RTT is that the ping round trip time does not provide any indication of the cache server load and the speed of the cache server [38].

2.1.2 Minimum HTTP Request Latency

The response time of a HTTP request can be estimated by using the response time of previous HTTP requests sent to the same cache server. This method assumes that the

HTTP request response times are stable within a short period of time. HTTP Request Latency, a substitute of HTTP Request Response Time, is measured from sending the request until the first byte of a response is received and is therefore independent of the size of objects. Unlike RTT, the HTTP request latency reflects not only the actual network load on the route between the client and the server, but also the server workload and speed.

In the HTTP request latency algorithm, a client sends requests to the server with the lowest median HTTP request latency in prior transfers. The problem with this approach is that prior latency computations may not successfully estimate the current response time because network load and server load change all the time.

2.1.3 Content-based Selection

Content-based routing decisions are made at Layer 5 of the OSI protocol hierarchies. Switch-based redirectors may operate at Layer 4 (network layer) or Layer 5 and above (Application layer) [11]. Redirectors providing Layer 4 services use TCP or UDP transport layer information, and can be configured to direct all traffic with particular destination TCP ports to a particular network port. Layer 5 switches use information found in the payload of HTTP request header packets. In order to obtain the HTTP request header, a Layer 5 switch sends a TCP SYN_ACK message to the client and tricks it into believing that there is a TCP connection established between the client and the server. The client then sends the HTTP request to the Layer 5 switch. A Layer 5 switch [7] makes the routing decision based on the availability at cache servers and type of the content.

2.1.4 Workload-based Selection

Some switches can intelligently redirect HTTP requests to lightly loaded cache servers. For example, a switch can determine which server gets the next connection by keeping a record of how many connections each server is currently providing, then the server with fewest connections gets the next request.

However, such switches as Cisco CSS 11000 [14] series switches, select servers based on server load and number of connections, and are blind to network latency. Therefore, they are only suitable for local cache clusters.

2.1.5 LB_L5 Selection

LB_L5 [30] is a fully distributed web-caching scheme that extends the capabilities of Layer 5 switching to improve the response time and to balance cache server workload. In LB_L5, a Layer 5 switch selects the best server based on cache content, cache server workload, network load and the HTTP header information. If the network latency between a cache server that stores the object and the Layer 5 switch is smaller than some threshold, then that cache server is considered as a candidate for access, and the Layer 5 switch uses load balancing algorithms to choose the best server to retrieve the object. The drawback of this approach is that it is difficult to set the threshold value and cannot guarantee the minimum request response time.

2.1.6 Minimum Response Time

To improve the LB_L5 selection algorithm, the Minimum Response Time (MRT) algorithm [45] aims to achieve the minimum expected response time of requests. In this scheme, cache server selection is based on the expected value of response time for both HTTP request cache-hits and HTTP request cache-misses. MRT selects the cache server with the minimum expected value of a HTTP request response time by considering three factors:

1. P_{cs_miss}, the probability that a predicted cache-hit HTTP request is a cache-miss on a cache server.
2. The delay components for a cache-miss HTTP request, T_{cs_miss}.,
3. The delay components for a cache-hit HTTP request, T_{cs_hit}.

MRT estimates the expected response time of a request as follows:

$$E(RT)_{cs} = P_{cs\_miss} * T_{cs\_miss} + (1-P_{cs\_miss}) * T_{cs\_hit}$$

MRT assumes that the size of requested objects is not significantly large and that all requests are equivalent.

2.2 Differentiated Service

Quality of Service (QoS) [15] aims to provide better service to selected traffic, facilitate priority by providing dedicated bandwidth, controlled jitter and latency (required by some real-time and interactive traffic), and improved loss characteristics.

Differentiated Services (DiffServ) is a means of providing QoS [25] by enforcing the aggregate traffic contracts between domains and ensuring that new sources of marked packets[i] do not cause traffic profiles to be violated. The idea behind the DiffServ architecture is based on aggregation of flows. Service differentiation is provided through service classes rather than providing per-flow QoS management. Core routers do not maintain per-flow states, they forward packets based on Per Hop Behaviors (PHB) encoded in the IP header's DiffServ Code Point (DSCP) field, which specifies the class of service for each packet, and can be used to provide the appropriate expedited handling by nodes throughout the network.

2.2.1 Differentiated Service Architecture

The DiffServ architecture is based on a simple model where incoming traffic is classified and assigned to different behavior aggregates (BA). The behavior aggregate information is encoded in a DSCP's. Within the core of the network, packets are processed according to their DSCP's.

A DiffServ structure contains one or more DiffServ Domains, as illustrated in Figure 2.2. A DiffServ domain consists of a contiguous set of DiffServ nodes that are subject to a common service provisioning policy and a set of PHB groups implemented on each node. The boundary of a DiffServ domain consists of DiffServ boundary nodes, which classify and mark incoming traffic, which is handled using PHB supported within the domain. Nodes within the DiffServ domain select the forwarding behavior for packets based on their DSCP, and map that value to one of the supported PHBs.

[i] Packets are marked at the IP-layer packet using the DS field at DiffServ domain boundaries.

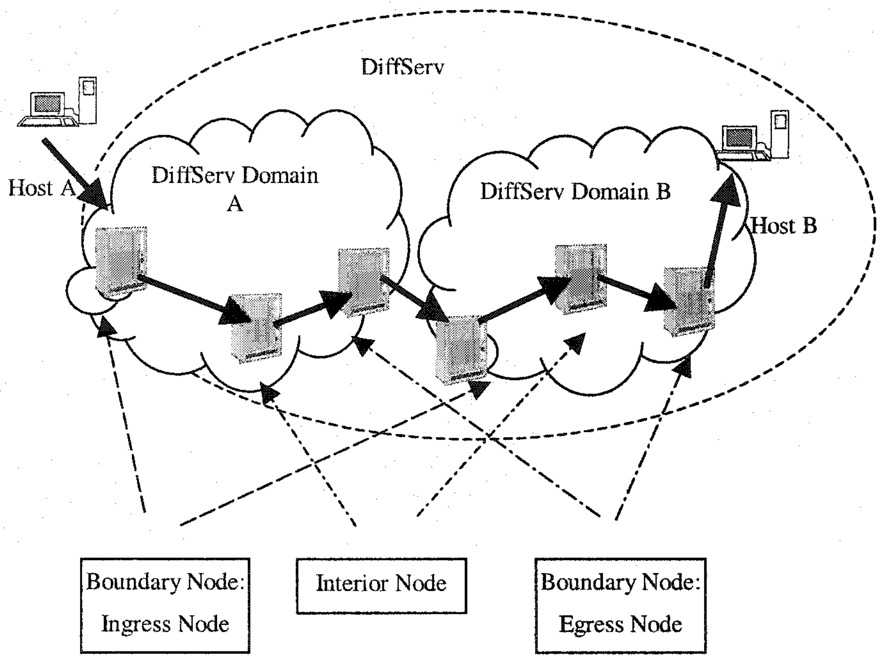


Figure 2.2 DiffServ Architecture [23]

DiffServ boundary nodes may perform traffic conditioning functions as defined by a traffic conditioning agreement (TCA) between their DiffServ domains and peering domains they connect with.

The TCA may specify packet classification and re-marking rules and may also specify traffic profiles and actions to traffic streams, which are in- or out-of-profile. The packet classification policy identifies the subset of traffic, which may receive a differentiated service by being conditioned and/or mapped to one or more behavior aggregates within the DiffServ domain. After classification, a traffic conditioner performs metering, shaping, policing and/or re-marking to ensure that the traffic entering the DiffServ domain conforms to the rules specified in the TCA, in accordance with the domain's service provisioning policy.

After being classified and conditioned, a packet is assigned a per-hop behavior (PHB). A PHB is a description of the externally observable forwarding behavior of a DiffServ node applied to a particular DiffServ behavior aggregate. A node allocates resources to behavior aggregates according to PHB, which may be specified in terms of their resource (e.g., buffer or bandwidth), priority relative to other PHBs, or in terms of their relative observable traffic characteristics (e.g., delay or loss). IETF has defined the Expedited Forwarding PHB (EF PHB) [22] and Assured Forwarding PHB (AF PHB) [24] to implement premium service and assured services, respectively. The premium service is a low loss, low latency, low jitter, assured bandwidth, end-to-end service through DS domains, and strictly enforces traffic layer. The assured service provides several classes with different layers of drop priority, but with fuzzy service guarantees.

2.2.2 Differentiated Service in Web Caching

With the dramatic increase of web traffic on the Internet, proper web caching schemes become crucial to providing adequate service to clients. Since caching resources such as cache size and disk-bandwidth are not infinite, they need to be allocated in an efficient manner. Current web caching systems that treat all client requests alike regardless of client importance and availability of resources may not make the best use of their finite resources in a heterogeneous environment. In the future, Internet services will be priced and differentiated service support will be in place. So caching policies should be able to adapt not only to clients' access patterns but also to the importance or DiffServ class of clients or contents. For example, an

ISP can have agreements with preferred content providers to give their site better service for a negotiated price. To enable such differentiation in a web caching system, we can consider implementing DiffServ at web servers, cache servers, or switches.

DiffServ can be achieved on the server side at either the system layer or Application layer. In general, at the system layer, different layers of resources including CPU, disk, network bandwidth, and physical memory are provided to different classes; at the Application layer, scheduling algorithms are applied to requests.

One way to incorporate DiffServ at the system level is Reservation Domains [8]. A reservation domain is a collection of processes and corresponding resource reservations. A computer system may run several reservation domains and provide several types of resources (e.g., CPU, disk, network, physical memory), which are reserved and scheduled independently. The processes that belong to a particular reservation domain are guaranteed to receive at least their reserved portions of the domain's associated resources.

Reservation Domains have the following benefits:

1. They provide QoS guarantees even when the system is overloaded. A reservation domain is similar to a smaller, dedicated machine, so application programs need not be rewritten to use real-time services for delivering QoS in a shared environment.

2. They allow division of resources according to a policy. For example, two reservation domains may each reserve half the CPU, although one of them contains more processes than the other, and all processes are CPU bound.

Service Differentiation can also be implemented at the kernel-level. A scheme proposed by Almeida, Dabu, Manikutty and Cao [3] directly maps the user-level request priority to a kernel-level process priority. In this approach, the Apache HTTP server [5] is modified to have each HTTP process call the kernel to record the priority of the current request it is handling. The kernel is responsible for mapping this priority into the process priority and executing the Sleep policy to decide if the process should proceed or block. When a process finishes handling a request, it calls the kernel again to release its priority and execute the Wakeup policy. The kernel first decides the priority level of the process to be unblocked, and then it chooses the process that blocked the earliest and originally was running on that priority level. When choosing a process to sleep, the kernel picks the process that started running the latest among all of those running on the lower priority.

For multimedia service, Chandra, Ellis and Vahdat [10] propose a Transcoding scheme that controls the consumed bandwidth for the different classes by proportionally reducing the image quality until the consumed bandwidth equals twice the target bandwidth, at which point further requests are denied. For preferred clients, the server reduces the image quality factor of the images served at a rate that is proportional to the overall target bandwidth. For the rest of the clients, the server reduces the image quality factor of the images served at a rate that is proportional to the leftover bandwidth.

At the Application layer, application specific information is maintained and utilized. For example, admission control algorithms can bind application data losses below a specified value; users can dynamically choose their level of network quality based on the resource cost, or select a lower quality multimedia object on a slower (cheaper) network in order to improve the access latency [10].

At a cache server, the usual approaches to providing a differentiated caching service is to consider how to allocate cache size or how to replace objects. Zhou and Philbin [44] propose Multi-Level LRU (ML-LRU), a replacement algorithm for performance differentiation in web caches. A cache using this algorithm maintains multiple queues, corresponding to different classes of objects. With service differentiation, the expected lifetime of premium objects is increased. In this algorithm, each queue employs simple LRU for object replacement within its own level. In addition, the queues are interconnected such that an object evicted from higher level is inserted into the head of the adjacent lower level. Upon a cache hit, an object is transferred back to the head of its original level.

Lu, Saxena and Abdelzaher [31] propose a control-theoretical approach to increase the hit rates of higher-class requests and thus increasing the client-perceived performance of these requests. The approach assumes N classes of users/traffic, the average delay of class j is Dj, and Kj is the specified weight for class j. To achieve the objective: $D1{:}D2{:} \ldots{:} DN = K1{:}K2{:} \ldots{:} KN$, an error $ej = Kj/(K1+K2 + \ldots +KN) - Dj/(D1+D2 + \ldots + DN)$ for class j is measured per feedback loop. Then resource allocation is adjusted based on the error. The approach addresses the problem as one of controller design and leverages principles of digital control theory to achieve an efficient solution. Results suggest that the approach results in a very good controller design compared to manual parameter tuning approaches.

2.3 Caching Streaming Media Content

The increasing popularity of multimedia content on the Internet has stimulated the emergence of streaming media applications. Digitization of video and audio yields a sequence of frames or samples that are referred to as streams [21]. When media content is transmitted in the steaming mode rather than download mode, it is played out while parts of the content are being received and decoded. This enables the client to initiate display of data with only small start-up latency and without waiting for the entire file to be downloaded. Due to the real-time nature of media data, streaming mode typically has bandwidth, delay and loss requirements and long duration. Caching streaming media objects becomes especially attractive due to the static nature of the content, predictable sequential nature of accesses, and the high network resource requirements.

2.3.1 Caching of Streaming Media Content

Caching streaming media [32] files has following characteristics [4]. First, real-time multimedia has timing constraints, which means, audio and video data must be played out continuously at a specific rate. Web caches need to reserve cache bandwidth for each media file, so media files stored in the cache require both bandwidth and space occupation. For this reason, only a limited number of media files can be guaranteed QoS. Second, compared with other Internet applications such as email and web browsing, streaming media requires high data rates and consumes significant bandwidth over long periods of time. Third, traffic generated by streaming media tends to be bursty and is sensitive to delay. Finally, steaming media files are very large in size; so they require significant storage that may be tens of megabytes to tens of gigabytes. For this reason, cached content must be stored on disks, not in memory caches.

To meet these requirements, some streaming media caching algorithms have been proposed and evaluated, such as Least Relative Benefit (LRB) [43], Resource-Based Caching (RBC) [40] and Prefix Caching [39] [29]. RBC manages the heterogeneous requirements of multiple data types by considering disk bandwidth as well as disk storage capacity, and caches a mixture of intervals and full files that have the greatest caching gain. LRB modifies the RBC policy to cache prefixes to facilitate further savings on network latency. Prefix Caching considers disk-bandwidth, and caches prefixes in cache servers. Prefix Caching is not as complicated as LRB or RBC, but satisfies our research requirements, and hence is used in our system.

2.3.2 Prefix Caching

Instead of caching entire audio or video streams (which may be quite large), the proxy stores a prefix [39][29] consisting of the initial frames of each clip. Upon receiving a request for the stream, the proxy immediately initiates transmission to the client, while simultaneously requesting the remaining frames from the server. In addition to hiding the latency between the server and the proxy, storing the prefix of the stream aids the proxy in performing work-ahead smoothing into the client playback buffer. By transmitting large frames in advance of each burst, work-ahead smoothing substantially reduces the peak and variability of the network resource requirements along the path from the proxy to the client.

2.4 Summary

In this chapter, we first introduced distributed web caching and switch-based cache selection algorithms. Switch-based selection algorithms have the following advantages. First, a switch is optimized for examining and processing packets, so there is minimal impact on non-Web traffic. Second, removing the packet examination, server selection, network address translation and routing functions from the cache server frees up CPU cycles for serving Web pages. Third, using a switch redirector that is separate from the cache servers allows the client load to be dynamically spread over multiple cache servers, which, in turn, can reduce response time. Further, redundant redirectors can be deployed, eliminating any single point of failure in the system. For above reasons, we concentrate the research effort on switch-based web caching systems.

We then introduced QoS and Differentiated Service (DiffServ) in web caching applications. DiffServ is a trend in Internet applications. For one thing, Internet infrastructure resources are limited; CPU, memory, disk, network bandwidth, etc. can not be enough to satisfy the rapidly growing demand of clients and applications. For another, Internet customers who pay more for their service need relatively better service, and some applications themselves have DiffServ requirements. When achieving DiffServ in web caching, we can consider locations, such as web server, proxy, switch, or client. We can also consider different levels at which to achieve DiffServ, such as system level, or application level. As DiffServ is a systematic task from web server to client, it should be addressed anywhere along this data path. Up to now, proposals have been made to achieve DiffServ at Network layer and Application layer. Our research is to deploy DiffServ in an integrated web-based system, from Application layer to Network layer. As the first step, we propose a DiffServ scheme at the Application layer, which takes advantage of DiffServ at the lower two layers.

Multimedia is another major concern of our research, as it is becoming increasingly popular and accounts for around 50% of data available on web servers. To achieve DiffServ, we select multimedia content as one class of data requested. However, due to the large size and real-time requirements of such content, a different caching algorithm is required. Research shows that Prefix Caching is an effective way of caching streaming multimedia content. In our research, we use prefix caching for streaming media objects.

Chapter 3 Switching-based Differentiated Service Caching (SDSC)

In this chapter, we present a scheme for Switching-based Differentiated Service Caching (SDSC) that uses the client request header, cache server content, cache server disk bandwidth, cache server workload, web server workload and link delay to effectively forward requests arriving at a switch to the most appropriate cache server or web server. The goals of SDSC are:

1. to classify Internet requests into different classes;
2. to provide differentiated DiffServ for different classes, including streaming multimedia content; and
3. to balance cache server workload.

3.1 Overview of SDSC

SDSC extends the LB_L5 transparent Web Caching Scheme [30] to include DiffServ provisions. The system structure is shown in Figure 3.1, where enterprise networks are connected with each other to achieve cache content sharing among the networks. Each network can be a local area network (LAN), and can belong to a same organization. SDSC is transparent, which means that each switch (thus the enterprise network where the switch is in) has the content information of all the cache servers it connects with. A client cluster is connected to its local L5 switch, which in turn connects with all cache servers and the web server. A cache server connects with all switches and the web server. There is no direct connection between cache servers. Cache sharing is achieved through the L5 switch.

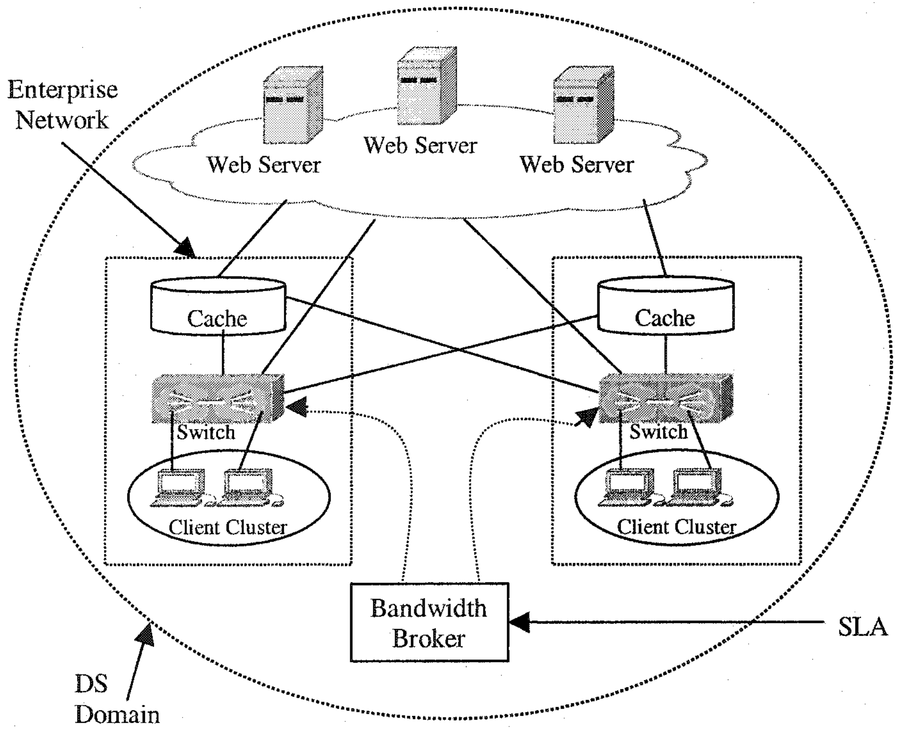


Figure 3.1 Structure of Switching-based DiffServ

SDSC consists of two components: client request classification and service provisioning, both of which are achieved on switches. L5 switches check HTTP request content, categorize requests into three classes: Streaming-object Class (SC), Real-time Assured Class (RC) and Best-effort Class (BC). For each class, a switch applies a corresponding routing algorithm. For SC, we introduce a new web-caching algorithm that extends prefix caching, namely, Disk-Bandwidth Routing (DBR).

DBR determines which cache server with enough disk-bandwidth can return the first byte of the requested SC object with minimum response time and within the specified delay bound. It then forwards the request to that cache server. If the cache server does not contain requested object in whole, the switch receives the remainder from the web server. For RC, Minimum Response Time (MRT) [45], which guarantees the minimum response time, is used. For BC, Workload Routing (WR) is used. It chooses a cache server with the lowest workload, thus balancing the workload of all cache servers.

DBR is significantly different from MRT and WR in three aspects:

1. a switch considers disk-bandwidth consumption of cache servers in DBR;
2. for large streaming media objects, the cache server holds the prefix and the remainder is transmitted directly from the web server; and finally,
3. for a request to be accepted, the first byte must be delivered within a specified Delay Bound (DB).

When measuring performance, there are some further differences, such as:

1. SC requests are measured with the response time of the first byte of the requested object, while RC and BC objects are measured with the response time to receive the last byte;
2. to guarantee the response time is within DB, we measure the acceptance rate (AR) for DBR, since requests that cannot be satisfied within this bound must be dropped.

Basically, SC has a higher priority than RC, which in turn has a higher priority than BC. The priorities are manifested in three ways:

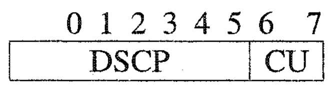
1. the processing time for SC objects on caches or the web server is faster that for RC, which in turn is faster than that for BC;
2. the link latency of SC packets is lower than that of RC, which in turn is lower than that of BC (This reflects the fact that packets carrying SC traffic have expedited PHB);
3. the average response time of RC requests should be smaller than that of BC even if the link delay and processing time are the same for both RC and BC.

In SDSC, we make the following assumptions:

1. the disk-bandwidth for processing a SC request at a cache server is the same and constant for all SC objects;
2. the objective of the system is to minimize service delay. We define service delay for SC requests to be the delay before the first byte of an object is received by the client;
3. streaming media objects are transmitted as constant bit-rate (CBR) traffic;
4. the web server is not a bottleneck for processing requests and has sufficient storage space and disk-bandwidth.

3.1.1 Client Request Classification

The DiffServ architecture uses the DS byte in the IP field to reflect the classification. Six bits of the DS field are used as a codepoint (DSCP) to select the PHB a packet experiences at each node. The DS field structure is presented in Figure 3.2:



DSCP: differentiated services codepoint CU: currently unused

Figure 3.2 DS Field Structure

DS-compliant nodes select PHBs by matching against the entire 6-bit DSCP field. The DSCP field is defined as an unstructured field to facilitate the definition of future per-hop behaviors.

At a switch, the client request classification component classifies Internet requests into one of SC, RC and BC and marks the DS byte of IP header accordingly, thus providing expedited forwarding PHB for premium service and assured forwarding PHB for assured service, as described in Chapter 2. This classification information is then carried until the request is completed.

3.1.2 Cache Content Representation

We use a Bloom Filter [17][34] (see Appendix A) to represent cache server content, including SC prefixes as used in the cache digest scheme [37]. A Bloom Filter is an array of bits. To represent an object in a Bloom Filter, a fixed number of independent hash functions are computed from the object's key. The number of hash functions specifies how many bits are used to represent one object. Their hash values specify the bit positions that should be set to1 in the Bloom Filter. All the bits are initially 0. When a key *a* (in our case, the URL of a document) is inserted or deleted, the counters $c(h_1(a))$, $c(h_2(a))$, ..., $c(h_k(a))$ are incremented or decremented accordingly.

When a count changes from 0 to 1, the corresponding bit is turned on. When a count changes from 1 to 0 the corresponding bit is turned off.

In SDSC, switches and caches use the same hash functions and bit arrays. Therefore, a URL is represented in the same way in all switches and cache servers. To make switches aware of cache content, a cache server sends its content in the form of a Bloom Filter to the switches. A switch stores the content information for each cache server. When a switch needs to check whether a requested object is in a cache server, it uses the same set of hash functions for the request's URL and examines the corresponding bits in the cache server's Bloom Filter. If all of the matching bits are 1's then the requested object is assumed to be in that cache server. Otherwise the object is assumed to be not in the cache server.

An object could be placed in different cache servers. Therefore, the way a L5 switch chooses the best cache server to service the request is a key issue in the system. In the next section we describe how to find the most appropriate cache server for a request.

3.1.3 Cache Server Selection Principles

Generally speaking, a switch selects the cache server that can satisfy the request with minimum response time or the one that with minimum workload. A Layer 5 switch uses cache server workload and network latency to estimate the request response time for each cache.

In DBR, cache server selection for SC objects is based on the expected response time to receive the first byte of the requested object. DBR selects the cache server with the minimum expected response time to deliver the first byte. A switch uses 5 factors to determine the response time:

1. cache server content: if a cache doesn't contain the object, it will not be chosen;
2. cache server disk-bandwidth usage: if there is not enough disk-bandwidth in a cache server to process a SC request, then the cache server is not selected;
3. the workload of the cache server for SC requests: it influences the processing time; and
4. delay of SC link(s): link delay is part of the response time;
5. false hit probability: the probability that a predicted cache-hit HTTP request is a cache-miss on cache server.

In MRT, which is used for RC requests, cache server selection is also based on the expected response time in case of HTTP request cache-hit and in case of HTTP request cache-miss. MRT selects the cache server with the minimum expected request response time. The expected response time is:

$$E(RT)_{cs} = P_{cs\_miss} * T_{cs\_miss} + (1-P_{cs\_miss}) * T_{cs\_hit}$$

where P_{cs_miss}, is the probability that a predicted cache-hit HTTP request is a cache-miss on cache server; T_{cs_miss} is the delay component for a cache-miss HTTP request, and T_{cs_hit} is the delay component for a cache-hit HTTP request.

In Workload Routing, which is used for BC requests, cache server selection is based only on the content and current workload of a cache server. The workload includes all current requests. The L5 switch selects the cache server that contains the object with the minimum workload.

3.1.4 Processing Procedures for SC Requests

SC requests require that cache servers have enough disk-bandwidth, response time is within a delay bound, and the requested prefixes are in the cache servers. Therefore, in DBR, four cases exist where a SC request cannot be accepted:

1. no cache stores the requested object;
2. a cache server stores the object, but there is not enough disk-bandwidth in the cache server to handle the request;
3. a cache server contains the requested object and has enough disk-bandwidth to handle the request, but the minimum predicted response time for the request is larger than the delay bound; and
4. a false hit, that is, when a request is forwarded to a cache server, the cache server in fact does not have the object.

The protocol procedure for a cache-hit HTTP request for a SC object in DBR is illustrated in Figure 3.3. When a switch receives a HTTP_REQ (http request message), it uses its Bloom filters to determine the set of cache servers that potentially have the object. The switch next determines the subset of cache servers with enough disk-bandwidth and then selects the server from that subset with the minimum response time. If the minimum response time is within the delay bound (DB), then the following actions occur:

1. The switch forwards the request to the selected cache server by sending a TCP_SYN signal for a connection. The cache server sends a TCP_ACK to accept the connection. The time spent is the round trip time between the switch and the cache server.

2. The switch relays a HTTP_REQ to the cache server. The time required is estimated as half of the round trip time between the switch and the cache server.

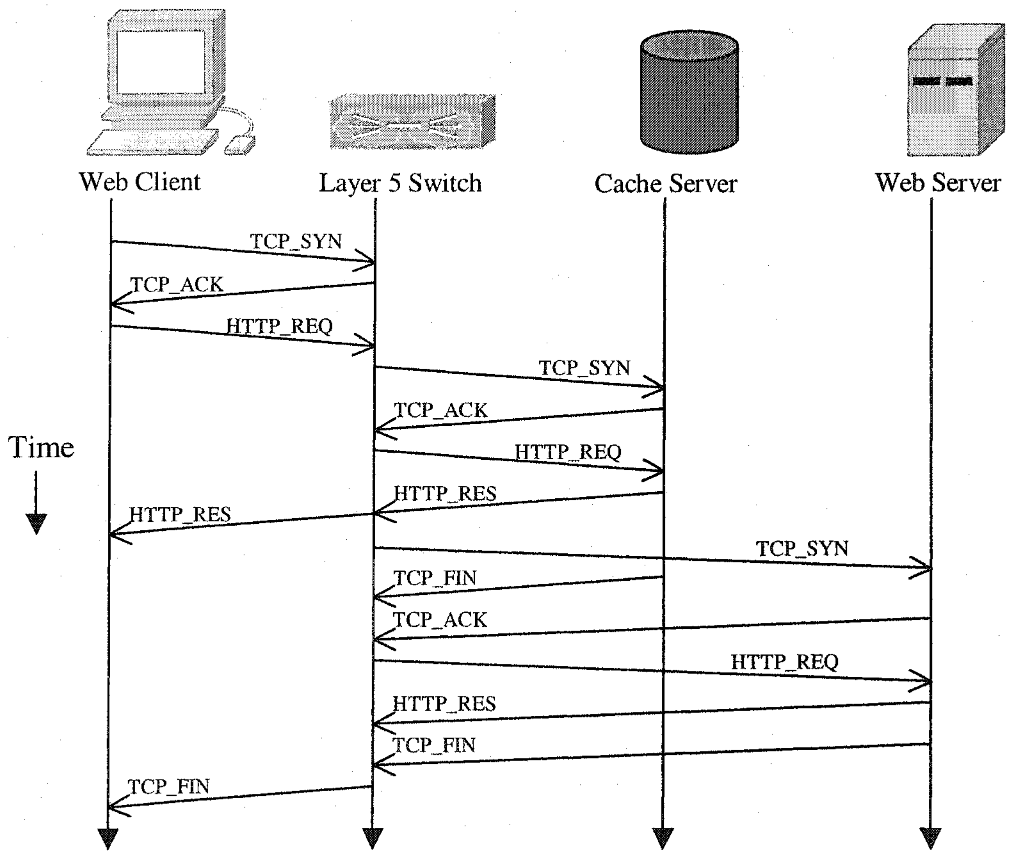


Figure 3.3 Cache Hit in Disk-Bandwidth Routing

3. The cache server processes the request. If there is not enough disk-bandwidth, the cache server returns a NULL object. If this is a true-hit and there is enough disk-

bandwidth, the cache server begins to occupy the disk-bandwidth. The time for the processing is proportional to the cache server's SC workload (definition and calculation of SC workload is presented in 3.1.6).

4. The cache server relays the requested prefix to the switch (HTTP_RES). The time spent is half of the round trip time between the cache server and the switch plus the time to transmit the whole prefix at the cache server. After transmitting the last byte of the prefix, the cache server releases the occupied disk-bandwidth.
5. From the perspective of the switch, the response time of the first byte of the prefix is the sum of the above components. The switch immediately sends the prefix to the client cluster. If the prefix is NULL or the total object size is not larger than the prefix size, then the switch sends TCP_FIN to the client after it receives TCP_FIN from the cache server. Otherwise, the switch initiates a connection with the web server by sending a TCP_SYN to the web server, which in turn returns a TCP_ACK.
6. The switch sends a HTTP_REQ to the web server for the remainder of the requested object, and the web server replies with the remainder (HTTP_RES). The time at the web server for processing the remainder of the requested object is proportional to the size of this remainder and the SC workload at the web server.
7. The web server sends the TCP_FIN to the switch, which in turn sends a TCP_FIN to the client cluster and ends the request procedure for the object.
8. From the perspective of the client, the response time of the first byte is calculated from the time the client sends a TCP_SYN until the client receives the HTTP_RES message; while the response time of the whole object is calculated from the time the client sends a TCP_SYN until receiving TCP_FIN.

The protocol followed for a cache-miss HTTP request for a SC object in DBR is illustrated in Figure 3.4. When a switch receives a HTTP_REQ from a client cluster, if it determines that no cache server stores the prefix, it sends a HTTP_RES and TCP_FIN to the client cluster. If there are some cache servers with enough disk-bandwidth to handle a SC request, the switch does the following:

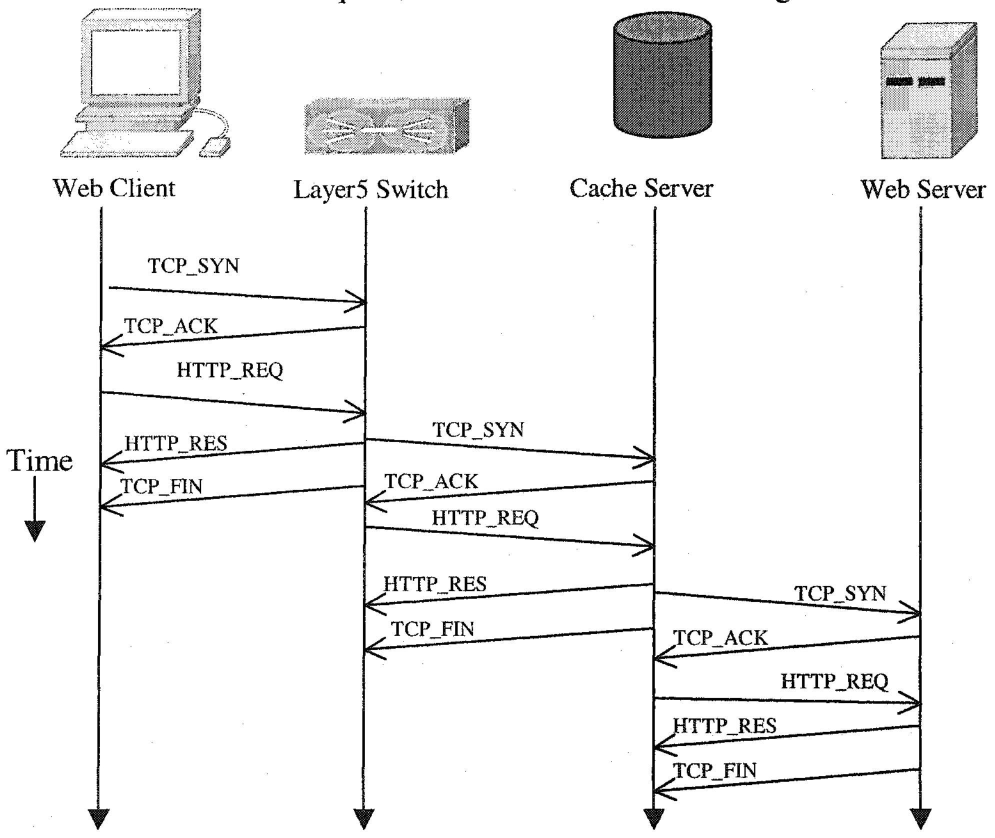


Figure 3.4 Cache Miss in Disk-Bandwidth Routing

1. It asks the cache server with most disk-bandwidth to get and cache the prefix from the web server. The switch sends a TCP_SYN to the cache server, which replies with TCP_ACK.
2. The switch sends HTTP_REQ to the cache server for the prefix of the object;

3. The cache server processes the request, and sends HTTP_RES with a NULL object and a TCP_FIN to the switch.

4. If the cache server has enough disk-bandwidth, then it sends a TCP_SYN to the web server for the prefix (and begins to occupy the disk-bandwidth for the request). The web server replies with TCP_ACK.

5. The cache server sends HTTP_REQ to the web server for the prefix, and the web server replies with HTTP_RES. The cache server stores a copy of the prefix. After the cache server receives a TCP_FIN from the web server, it releases the disk-bandwidth.

In the case of a false-hit, which is shown in Figure 3.5, the following events occur:

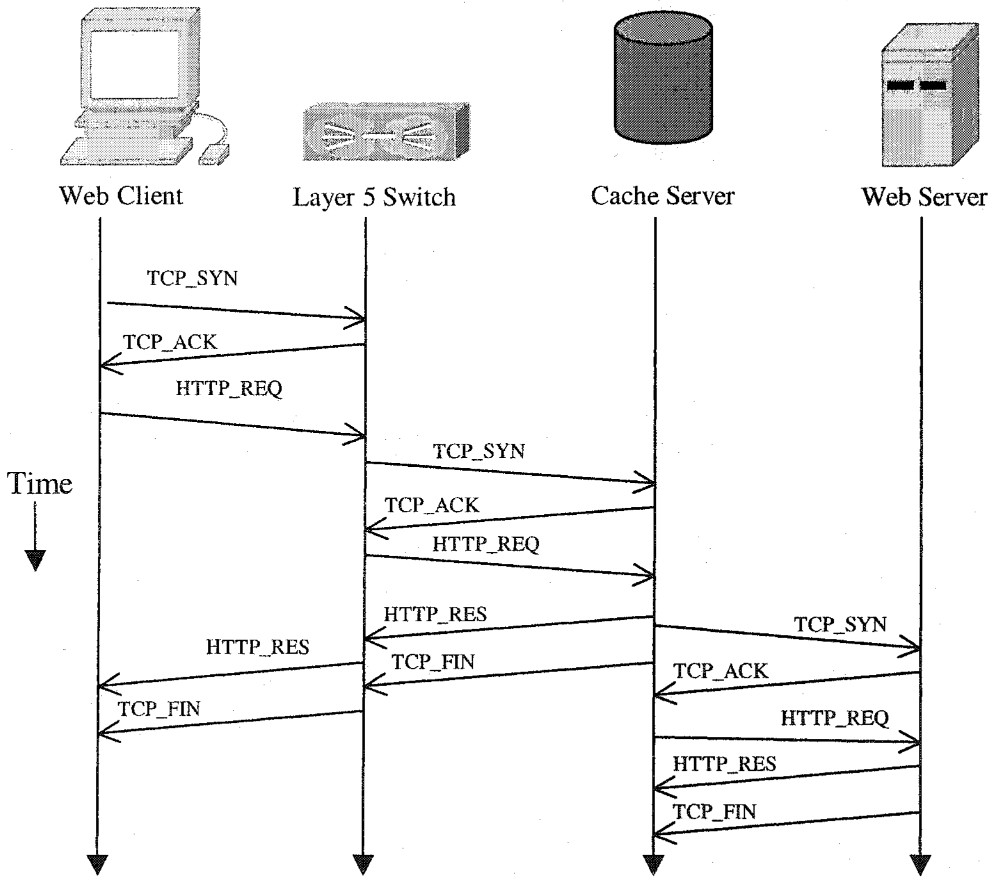


Figure 3.5 False Hit in Disk-Bandwidth Routing

1. A switch sends a TCP_SYN to the selected cache server, which returns a TCP_RES (TCP response message).

2. The switch sends a HTTP_REQ to the cache server, which sends a HTTP_RES with a NULL object, followed by TCP_FIN.

3. At the same time, the cache server checks if there is enough disk-bandwidth in the cache server. If there is not enough space, the cache server does nothing. Otherwise, it asks the web server to send the prefix.

4. The cache server sends TCP_SYN to initiate the connection with the web server and occupies the disk-bandwidth needed to receive the prefix. The web server replies with TCP_ACK.

5. The cache server sends HTTP_REQ for the prefix, and the web server sends back the prefix HTTP_RES, and the cache server stores a copy of the prefix.

6. The web server sends TCP_FIN after finishing sending the prefix, the cache server release the disk-bandwidth.

3.1.5 Protocols for RC Requests and for BC Requests

A switch uses MRT to forward an RC request, and uses the Workload Routing to forward a BC request. The cache server selection algorithms, processing queues, processing time, workload and link delays are different for the two schemes, and RC workload influences BC workload. The message exchange of both is the same. The cache-hit protocol is illustrated in Figure 3.6. After the switch receives TCP_ACK, then following actions occur:

1. The switch relays the HTTP request to the cache server. The time required is estimated to be half of the round trip time between the switch and the cache server.
2. The cache server processes the request. The time for processing is proportional to the cache server's workload.
3. The cache server relays the object to the switch. Again the time required is estimated to be half of the round trip time between the cache server and the switch.

A cache-hit request response time is equal to the sum of the above components.

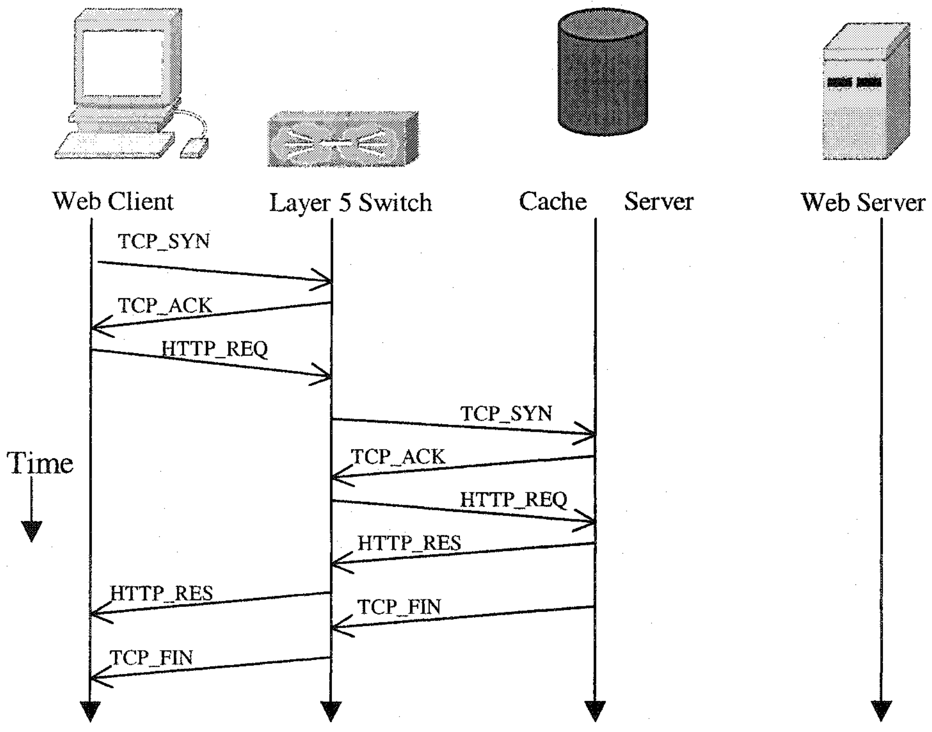


Figure 3.6 Cache Hit in MRT

Cache-miss and false-hit handling procedures are illustrated in Figure 3.7. After a switch receives a HTTP request from the Web client, it uses either MRT or Workload Routing.

1. The switch sends a TCP_SYN to a cache server, which sends back TCP_ACK. The time spent is the round trip time between the L5 Switch and the cache server.
2. The switch relays the HTTP request to the cache server. The time required is estimated to be half of the round trip time between the L5 Switch and the cache server.
3. The cache server processes the request for a time proportional to the cache server's workload.
4. Since this is a cache-miss or false-hit, the cache server sends a TCP_SYN to the web server, which sends back a TCP_ACK. The time required is the round trip time between the cache server and the web server.

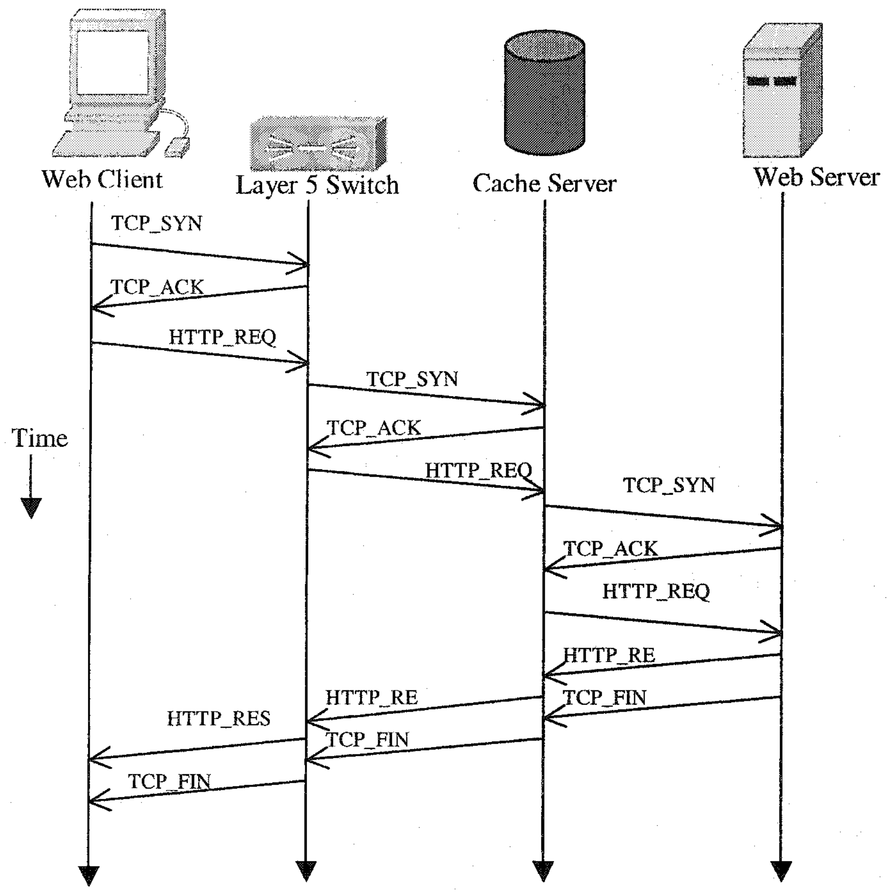


Figure 3.7 Cache-miss and False-hit in MRT

5. The cache server relays the HTTP request to the web server. The time required is estimated to be half of the round trip time between the cache server and the web server.

6. The web server processes the request for the time proportional to the web server workload.

7. The web server sends back the requested object to the cache server. The cache server stores a copy. The time spent is estimated to be half of the round trip time between the cache server and the web server.

8. The cache server relays the object to the switch. The time spent is estimated to be half of the round trip time between the cache server and the switch.

A cache-miss or false-hit request response time is equal to the sum of the above components.

3.1.6 Queuing at cache servers

SC requests, RC requests and BC requests are placed in separate queues at a cache server. In our DiffServ framework, resources are reserved for classes at each node that processes requests. In SDSC, we assume that requests belonging to three classes are in three separate queues, and the average processing time for class i, PT_i is constant.

The SC workload of a cache server is calculated as follows:

$$WL_{sc} = N_{sc} / MAX_{sc}$$

where N_{sc} is the number of SC requests at the cache server, and MAX_{sc} is the maximum allowed number of concurrent SC requests at the cache server. The values are periodically sent by the cache server to the switch.

The RC workload of a cache server is calculated as follows:

$$WL_{rc} = WL_{sc} + N_{rc} / MAX_{rc}$$

where N_{rc} is the number of RC requests at the cache server, and MAX_{rc} is the maximum number of concurrent RC requests at the cache server.

The BC workload of a cache server is calculated as follows:

$$WL_{bc} = WL_{rc} + N_{bc} / MAX_{bc}$$

where N_{bc} is the number of BC requests at the cache server, and MAX_{bc} is the maximum number of concurrent BC requests at the cache server.

After we get the SC, RC and BC workload of a cache server, we can get the processing time for a request of class i [36]:

$$T_i = WL_i * Max_i * PT_i \text{, } i \in \{SC, AC, BC\}$$

A switch can periodically get the three workloads of a cache server, and the maximum http request numbers of the classes, hence it can calculate and predict cache server processing time for a request.

3.2 Streaming-object Request Routing

DBR is based on a distributed architecture, where each cache server sends its SC content and number of SC objects to all switches. A switch periodically queries the workload of each cache server.

A Layer 5 switch in DBR makes a routing decision as follows:

If a cache server is predicted to store the requested prefix, the expected response time of the first byte of the prefix is calculated as:

$ES_{swi\_csj}(RT)=2*RTTS_{swi\_csj}+WLS_{csj}*MaxS_{csj}*PTS_{cs}$

where $RTTS_{swi\_csj}$ is the round trip time from switch i to cache server j for a SC object, WLS_{csj} is the SC workload at cache server j, $MaxS_{csj}$ is the maximum SC workload at cache server j, and PTS_{cs} is the average SC processing time at a cache server.

The switch selects the cache server with the minimum estimated response time.

To keep continuous and smooth transmission, the prefix size should be large enough such that the client receives the whole object without interruption. To achieve this, the first byte of the remainder transmitted from the web server should arrive at the switch before the last byte of the prefix transmitted from the cache server leaves the switch. The size of the prefix is calculated such that this condition is true.

To make the first byte of the remainder arrive from the web server before the last byte of the prefix leaves at a switch, $t_{remainder\_arrive} \leq t_{prefix\_leave}$ should be satisfied. Note that $t_{remainder\_arrive}$ and t_{prefix_leave} meet the following conditions:

$$t_{remainder\_arrive} \geq min(RTTS_{sw\_ws})*2+PTS_{ws}*min(WLS_{ws})*MaxS_{ws};$$

$$t_{prefix\_leave} = t_{prefix\_transmission}$$

$t_{prefix\_transmission}$ is calculated as follows:

$$t_{prefix\_transmission} = Size_{prefix} / Rate_{transmission};$$

Therefore, the size of a prefix is calculated as follows:

$$Size_{prefix} \geq Rate_{transmission} * (min(RTTS_{sw\_ws})*2 + PTS_{ws} * min(WLS_{ws}) * MaxS_{w})$$

3.3 Real-time and Best-effort Request Routing

A L5 switch uses MRT to make routing decisions for a RC request as follows:

If a cache server is predicted to store the requested object, the probability that the request is a cache-miss [34] is calculated as:

$$P_{cs\_miss} \approx Fp = (1- e^{-WDcs / Fcs})^{W}$$

Otherwise: $P_{cs\_miss} = 1$

The expected response time for a cache server is calculated as [45]:

$$ER_{swi\_csj}(RT)=P_{cs\_miss}*(2*RTTR_{swi\_csj}+WLR_{csj}*MaxR_{csj}*PTR_{cs}+2*RTTR_{csj\_ws}+ WLR_{ws}*MaxR_{ws}*PTR_{ws})+(1- P_{cs\_miss})*(2*RTTR_{swi\_csj}+WLR_{csi}*MaxR_{csi}*PTR_{cs})$$

where $P_{cs\_miss,}$ is the probability that a predicted cache-hit HTTP request is a cache-miss on cache server CS, $RTTR_{swi\_csj}$ is the round trip time for a RC object from switch i to cache server j, WLR_{csj} is the RC workload at cache server j, $MaxR_{csj}$ is the maximum RC workload at cache server j, PTR_{cs} is the average processing time of RC requests at a cache server, WLR_{ws} is the RC workload at the web server, $MaxR_{ws}$ is the maximum RC workload at the web server, and PTR_{ws} is the average RC processing time at the web server. The switch selects the cache server with the minimum estimated response time.

A L5 switch uses Workload Routing to make routing decisions for a BC request as follows:

1. Compare the BC workload of all cache servers.

2. Select the cache server with the minimum BC workload.

3.4 Information Exchange between switches and servers

We extend ICP (Internet Cache Protocol) [42] to exchange content, workload and disk-bandwidth information between a switch and a cache server. Similar messages are used by Liang [30] and Zou [45] . The ICP message format is as follows:

0	1	2	4
Opcode	Version	Message Length	
Request Number			
Options			
Option Data			
Sender Host Address			
Payload			

Figure 3.8 ICP Message Format

The following opcodes are used by SDSC:

- The ICP_UPDATE_CONTENT message is used by a cache server to periodically inform a switch about its cache content and the number of stored objects in the cache server. The Payload field carries the information: (1) a Bloom Filter, which represents the cache server contents, (2) the number of stored objects in the cache server.

- The ICP_QUERY_WORKLOAD_DBANDWIDTH message is used by a switch to periodically query the workload and disk-bandwidth usage of cache servers.

- The ICP_UPDATE_WORKLOAD_DBANDWIDTH message is used by a cache server to send its workload and disk-bandwidth information to the switch after it receives the ICP_QUERY_WORKLOAD_DBANDWIDTH message. It is also used by the web server to periodically send its workload information to the switches. The content of the Payload consists of two parts: (1) workload of the cache server and (2) disk-bandwidth usage of the cache server.

A L5 switch also needs an information table to maintain cache server information. An entry in the table is shown in Table 3.1.

IP Address Of cache server	Bloom Filter For SC	Bloom Filter For RC	Bloom Filter For BC	SC Count	RC Count	BC Count	SC Workload	RC Workload	BC Workload
SC Network Latency	RC Network Latency	BC Network Latency	Max SC Connection	Max RC Connection	Max BC Connection				

Table 3.1 Information Table at a Switch

3.5 Summary

In this chapter, we described Switching-based Differentiated Service Caching (SDSC) SDSC is latency-sensitive, achieves DiffServ for a number of request classes including streaming media requests, and achieves load balancing across cache servers. The client request classification component of SDSC classifies incoming HTTP requests into classes SC, RC or BC, and assigns PHBs of the classified requests at switches. The service provisioning component of SDSC uses three routing protocols to achieve DiffServ. DBR, MRT, and WR are the request processing protocols used for SC, RC and BC requests, respectively. DBR determines a cache server that has enough disk-bandwidth and that can return the prefix of the requested SC object with minimum response time and within the delay bound to forward the request. MRT, which is used for RC, chooses a cache server with minimum response time for the RC request. WR is used for BC and intended to balance the workload of cache servers. SDSC adjusts its routing decision dynamically based on request class, cache server disk-bandwidth usage, the cost of network latency and the cost of the workload to achieve minimum response time for SC and RC classes. Since SDSC extends LB_L5, it possesses the advantages of a distributed web caching, such as fault-tolerance (achieved by cache sharing) and routing efficiency (because a switch integrates most control and routing-related calculation functions).

Chapter 4 Performance Evaluation

In this chapter, we evaluate the performance of SDSC. We examine how different factors impact the performance of SC requests; we also study the effects of a variety of parameters on the system as a whole. Since SC class objects have a different caching mechanism and significantly larger content size, we do not compare DBR with MRT or WR. However, since changes in the SC cache size influences the cache size of RC and BC, and the workload of SC in a cache server or the web server influences that of RC and that of BC, we show the effects of these changes. On the other hand, RC objects and BC objects are similar in terms of caching mechanism and object size, so we can compare MRT and WR.

Section 4.1 describes the simulation model, which includes the network model, the network latency model, the workload model and the simulation parameter settings. Section 4.2 describes how DBR, MRT and WR are implemented in our simulation. Section 4.3 presents the performance metrics that we use to evaluate the performance of the different algorithms. Section 4 investigates the effects of cache size allocation, disk-bandwidth, delay bound, ratio of SC, RC and BC requests, client cluster request intensity, and network latency factors on the performance of DBR, MRT and WR. Section 4.5 summarizes the chapter.

4.1 Simulation Model

The simulation model consists of a system model, a network model, a network latency model and a workload model.

4.1.1 System Model

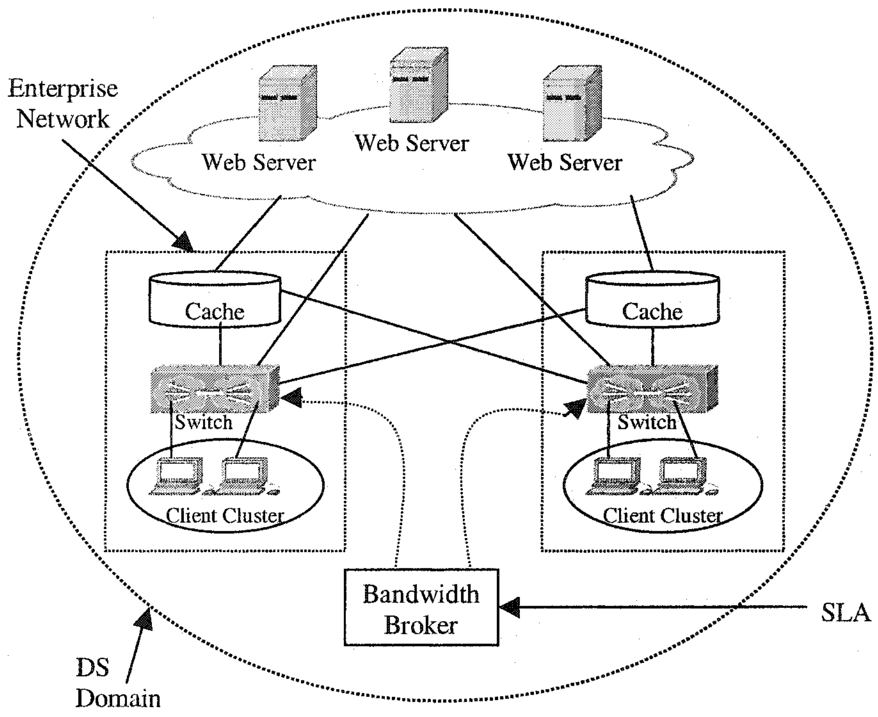


Figure 4.1 System Model of Switching-based Web Caching

The System model of SDSC is shown in Figure 4.1. The model simulates a distributed switching-based web caching system, where a client cluster is connected to a local L5 switch, which in turn is connected to all cache servers and the web servers. There is no direct connection between cache servers or between a client cluster and a cache server. Cache sharing is achieved through the L5 switch. Since we assume the web server is not a bottleneck, we use a web server cluster (the web server) to simulate the web server. A bandwidth broker in the DS domain is responsible for marking and shaping requests, which is part of the switch functions.

4.1.2 Network Latency Model

The network latency model, which is similar to the one used by Zou et al [45], is shown in Figure 4.2. We use a distance metric to reflect the costs of transferring data between any two nodes. This method is introduced by M. Rabinovich [18]. The cost of transferring data between two nodes is proportional to the distance between the pair of nodes. Since SDSC is delay-sensitive, the cost in the system is time, that is, the network latency between two nodes is proportional to the distance between them. The network latency model is a symmetric architecture with n client clusters, n switches and n cache servers, and a web server cluster. In our simulation, n is 4.

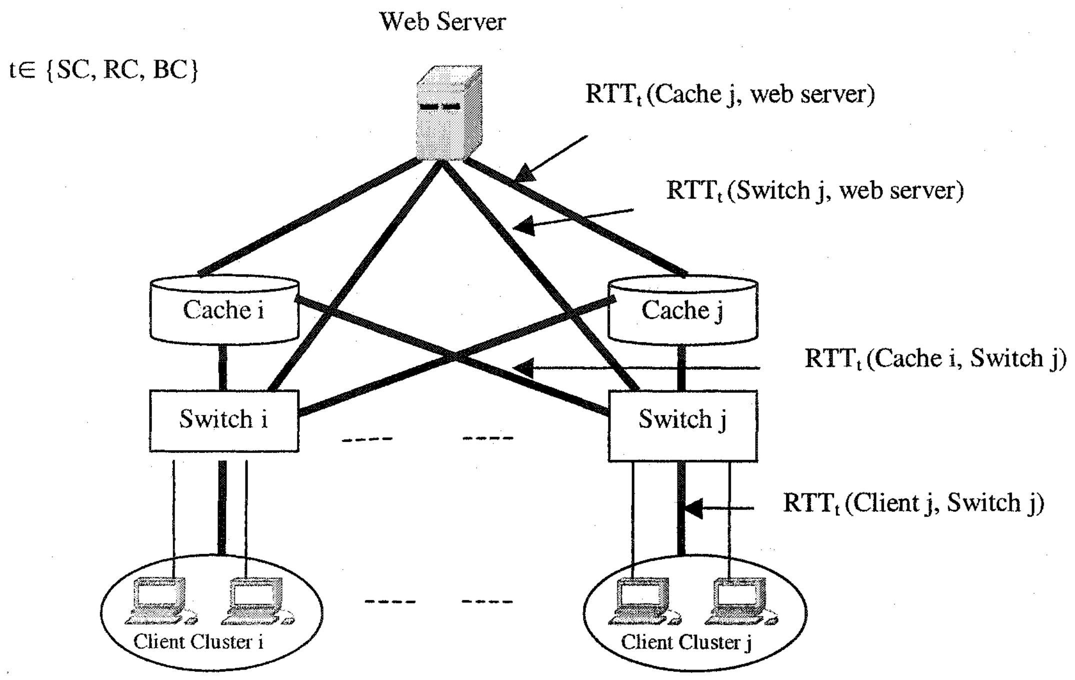


Figure 4.2 Network Latency Model [45]

There are different PHBs for SC, RC and BC, namely, SC should be provided with expedited service, RC with assured service, and BC with best-effort service. Therefore, between any two nodes, there are three levels of link delay. Quantitatively, the basic link delay for SC is shorter than that for RC, which in turn is shorter than that for BC. The network latency between node i and node j is calculated as follows:

$$NetworkLatency\ (i, j) = Distance\ (i, j) * LatencyFactor_k,\ k \in \{SC, RC, BC\}$$

$$Distance\ (i, j) = |i-j|$$

where *NetworkLatency(i,j)* is the time (in milliseconds) to transfer data from node i to node j, and *LatencyFactor* is the time (in milliseconds) to transfer data for one unit of distance.

4.1.3 Workload Model

There are usually two ways of generating workload: random number generation and traces [2][7]. We use traces for the simulation because traces can account for client access patterns and the requested content. The trace we use to generate HTTP requests is a publicly available proxy trace from the NLANR [28]. Each entry in the trace has nine fields, five of which are used in the simulation. The five fields are:

1. *Timestamp*: specifies the time when the client generates the HTTP request. The format is "Unix time" with millisecond resolution.
2. *Client Address*: The IP address of the client cluster.
3. *Size*: The number of bytes transferred from the proxy to the client.
4. *URL*: The uniform resource locator, which is a character string describing the location and access method of a resource on the Internet.
5. *Content-Type*: type of requested object (used for packet marking).

The average workload of a server is calculated as follows:

$$AverageWorkload = \frac{\text{Average Num of TCP Connections Per Second}}{\text{Maximum Num of TCP Connections of the Server Per Second}}$$

The average number of TCP connections is calculated as follows:

$$AvgTCPNum_t = (1-W_q) * AvgTCPNum_{t-1} + W_q*TCPNum_t$$

where $AvgTCPNum_t$ is the average number of TCP connections per second at time t, $TCPNum$ is the active number of TCP connections, W_q is a weight factor, $0 < W_q < 1$.

After some initial testing, a value of $W_q = 0.02$ has been chosen in our simulation. The average number of TCP connections in our simulation is therefore calculated as:

$AvgTCPNum_t = (1- W_q) * AvgTCPNum_{t-1} + W_q * TCPNum_t$

4.1.4 Simulation Parameter Setting

The parameter settings used in the simulation are summarized in tables 4.1 ~ 4.4. Parameter settings are similar to those used in [30] and [45]. Parameter values listed in the tables are average time. For instance, $RTTS_{cc\text{-}sw}$ is the average round trip time between a Client Cluster and a switch for SC. In simulations, the real round trip time is related to the workload of cache servers. We don't consider the impact of traffic on the network latencies. Moreover, queuing is not part of processing values.

Name	Meaning	Value (milliseconds)
$RTTS_{cc\text{-}sw}$	The round trip time between a Client Cluster and a switch for SC	65
$RTTR_{cc\text{-}sw}$	The round trip time between a Web client and a switch for RC	130
$RTTB_{cc\text{-}sw}$	The round trip time between a Client Cluster and a switch for BC	156
$RTTS_{sw\text{-}cs}$	The round trip time between a switch and a Cache Server for SC	0~200
$RTTR_{sw\text{-}cs}$	The round trip time between a switch and a Cache Server for RC	0~400
$RTTB_{sw\text{-}cs}$	The round trip time between a switch and a Cache Server for	0~480

	RC	
$RTTS_{sw\text{-}ws}$	The round trip time between a switch and the Web server	150
$RTTR_{sw\text{-}ws}$	The round trip time between a switch and the Web server	300
$RTTB_{sw\text{-}ws}$	The round trip time between a switch and the Web server	360
$RTTS_{cs\text{-}ws}$	The round trip time between a Cache server and the Web server	150
$RTTR_{cs\text{-}ws}$	The round trip time between a Cache server and the Web server	300
$RTTB_{cs\text{-}ws}$	The round trip time between a Cache server and the Web server	360

Table 4.1 Parameters for Network Links

Name	Meaning	Value (milliseconds)
QueryWorkload_Interval	The interval between two QueryWorkload msgs	1000
TCP_Splicing	The time it takes a switch port controller to translate TCP sequence number	0
RoutingS	The time it takes for a switch to make a routing decision for SC	10
RoutingR	The time it takes for a switch to make a routing decision for RC	10
RoutingB	The time it takes for a switch to make a routing decision for BC	10

Table 4.2 Parameters for Switches

Name	Meaning	Value (milliseconds)
WS_ProcessingS	The time it takes for SC at the Web Server between receiving a SC request and returning the first byte of the requested object	75
WS_ProcessingR	The time it takes for RC at the Web Server between receiving a RC request and returning the first byte of the requested object	150
WS_ProcessingB	The time it takes for BC at the Web Server between receiving a BC request and returning the first byte of the requested object	180
WS_ReplyS	The time it takes at the Web Server to send a SC object in memory to the requesting party	75
WS_ReplyR	The time it takes at the Web Server to send a RC object in memory to the requesting party	150
WS_ReplyB	The time it takes at the Web Server to send a BC object in memory to the requesting party	180
UpdateWorkload_Interval	The interval to send the updated workload	60*1000

Table 4.3 Parameters for Web Servers

Name	Meaning	Value
CS_DBandwidth	The total disk-bandwidth at a cache server for handling SC requests	64~256*1024 (bytes/sec)
Req_DBandwidth	The disk-bandwidth required to handle a SC	8*1024

	request at a cache server	(bytes/sec)
PrefixSize	The size of a SC prefix	8*1024*1024 (bytes)
CS_CacheSize	The total physical size of a Cache server for all SC, RC and BC objects	300*1024*1024 (bytes)
CS_SearchS	The time it takes at a Cache Server to search for a SC object in its cache	200 (milliseconds)
CS_SearchR	The time it takes at a Cache Server to search for a RC object in its cache	250 (milliseconds)
CS_SearchB	The time it takes at a Cache Server to search for a BC object in its cache	300 (milliseconds)
CS_SearchDigestS	The time it takes at a Cache Server to search for a SC object in its cache digests	80 (milliseconds)
CS_SearchDigestR	The time it takes at a Cache Server to search for a RC object in its cache digests	100 (milliseconds)
CS_SearchDigestB	The time it takes at a Cache Server to search for a BC object in its cache digests	120 (milliseconds)
CS_DiskAccessS	The time it takes at a Cache Server to retrieve a cached SC object from disk to memory	80 (milliseconds)
CS_DiskAccessR	The time it takes at a Cache Server to retrieve a cached RC object from disk to memory	100 (milliseconds)
CS_DiskAccessB	The time it takes at a Cache Server to retrieve a cached BC object from disk to memory	120 (milliseconds)
CS_ReplyS	The time it takes at a Cache Server to send an SC object in memory to the requesting party	120 (milliseconds)
CS_ReplyR	The time it takes at a Cache Server to send an RC object in memory to the requesting party	150 (milliseconds)
CS_ReplyB	The time it takes at a Cache Server to send an BC object in memory to the requesting party	180 (milliseconds)
CS_RelayS	The time it takes at a Cache Server to relay a SC response to the requesting party	40 (milliseconds)
CS_RelayR	The time it takes at a Cache Server to relay a RC response to the requesting party	50 (milliseconds)
CS_RelayB	The time it takes at a Cache Server to relay a BC response to the requesting party	60 (milliseconds)
CS_CacheDigest_Size	The Bloom Filter size of SC, RC or BC for a Cache Server	32*1024 (bytes)
CS_CacheDigest_Interval	The interval between two consecutive content_update msgs	1*60*1000 (milliseconds)

Table 4.4 Parameters for Cache Servers

4.2 DBR, MRT and WR Implementation

When a Layer 5 switch receives a HTTP response, it checks if the request is of SC, RC or BC.

1. If it is the prefix of a SC object, the switch checks if there is a remainder for the SC object. If there is, the switch forwards a request to the web server for the remainder; if there isn't, the switch ends processing for the request after sending the request to the client cluster.
2. If it is an RC or BC object, the switch forwards the object to the client cluster and ends processing for the request.

The pseudo code of the three algorithms is provided in Appendix B.

4.3 Performance Metrics

The following performance metrics are collected in our simulation experiments:

1. SC request acceptance rate (AR) is the percentage of SC requests that are satisfied within the delay bound (DB). The higher the AR value is, the better service SC requests receive.
2. Client request response time is the duration from when a client sends a TCP connection request to the time the client receives the TCP connection finished signal. It is affected by the workloads of cache servers and the web server, the network latency and false prediction. A client's perception of Web performance is based on the response time. The shorter the average response time is, the better the performance is.
3. Average cache server workload is the number of requests serviced during a given time interval. Cache server workload influences object retrieval times on the cache server, and the request response time (see Section 3.1.6). The average workload can be used to indicate the relative balance among the cache servers.

AR is a specific metric and only relevant for SC requests. The average response time is used to measure the performance for RC and BC requests. Our goal is to obtain a satisfactory AR for SC, to get minimum response time for RC, and to balance the workload of the cache server cluster.

4.4 Simulation Results

We begin to measure the performance metrics after all cache servers are filled up with prefixes and objects. The cache size allocation for each class is fixed during one simulation. We use raw NLANR traces as well as controlled traces to run the simulations. When using the raw trace, HTTP requests are generated at the time specified by the timestamp field in the trace file. When using controlled traces, the time interval between two consecutive HTTP requests is controlled, and the ratio of SC, RC and BC requests is also controlled, so as to vary the HTTP request intensity of each class. The experiments were run at a 90% confidence level with 5% confidence intervals.

4.4.1 Raw-trace Driven Simulations

The raw trace we use in our simulations is a NLANR trace obtained from cache servers - bo2_20020812 over a 24-hour period. Figure 4.3 shows the HTTP request intensity for one day, which is measured in number of HTTP requests in10-minute periods. The overall request intensity is relatively high from 8 am to 9 pm. Its peak intensity is 2226 requests per 10 minutes around 4 pm, and decreases significantly after 9 pm. The minimum intensity is 289 requests per 10 minutes around midnight.

The number of SC requests is lower and more uniform than the other types. It reaches the peak value of 186 at 5 pm. The number of RC requests reaches its peak of 957 at 11 pm, and the number of BC requests reaches its peak of 1604 around 5 pm. The BC line is most similar to the total request line since it accounts for the largest portion of requests. The overall ratio of the number of SC requests to RC requests to that of BC requests is 1: 4.6: 12.

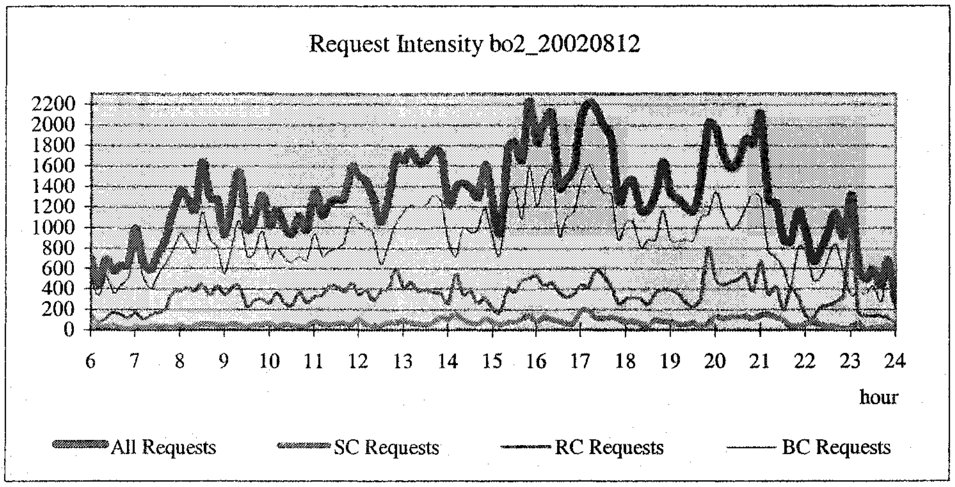


Figure 4.3 Request Intensity of Raw Trace

The request classification is based on the following:

1. SC requests are for audio and video objects.
2. RC requests are for e-commerce requests.
3. BC requests are for email and text files.

For the purpose of simulating cache cooperation among different network domains we use four different network domains for four different client clusters based on client IP addresses. The four domains are assigned to different client clusters. After 6 hours, the cache size is fully occupied and the system becomes stable. Our results are obtained for the following parameter settings: the latency factors for SC, RC and BC are 10ms, 20 ms and 24ms, respectively, DB is 800ms, DBandwidth is 80K (byte/sec). The acceptance rate from the 6th hour to the 24th hour is shown in Figure 4.4. We see that SC Acceptance is roughly inversely related to the SC request intensity in Figure 4.3, which means that when SC request intensity is high, the acceptance rate is low. There are some exceptions, such as around 5 pm, because the acceptance rate depends on other factors beside request intensity, including latency, delay bound (DB) disk-bandwidth (DBandwidth) and cache size. Generally speaking, the following is true:

1. A lower delay bound (DB) means a lower the acceptance rate (AR).
2. When disk-bandwidth is high, more SC requests can be handled simultaneously, so the acceptance rate is high.
3. When the latency factor is high, the link delay for SC objects is high, and DB may not be met.
4. If the cache size is large, then more prefixes can be cached. This increases the hit rate and means a higher acceptance rate.

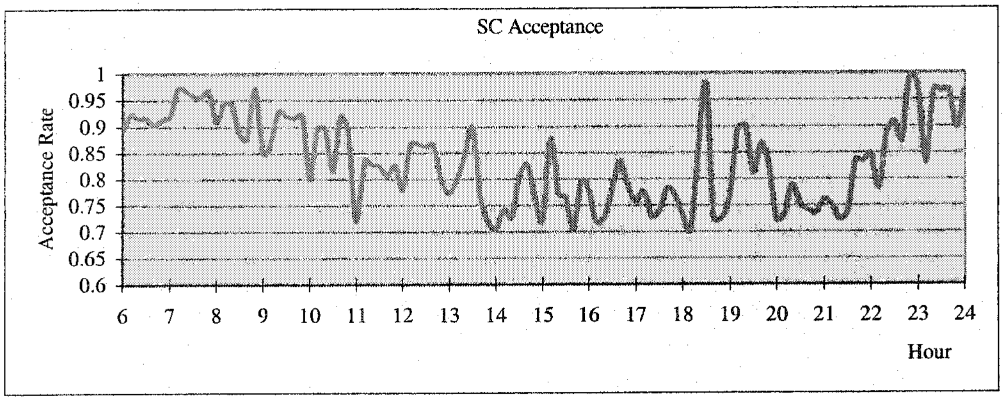


Figure 4.4 SC Acceptance Rate
DB=800ms, DBandwidth=80k, Latencyfactor=10ms, cache =80M

The average response time for RC and BC requests is shown in Figure 4.5. Results show that, and as expected, the response time of each class is proportional to its respective request intensity. Results also show that the average response time for RC is less than that for BC. This is due to a number of factors. First, MRT attempts to service RC requests with minimum response time. Second, in SDSC, the average processing time for RC is shorter than that for BC, cache size in a cache server for RC is more than that for BC, and link delay for RC is less than that for BC. Besides, BC request intensity is higher than RC request intensity. The combined effect of these factors leads to a lower average RC response time than that for BC. We will later show a case comparing the MRT and the Workload schemes under the same conditions.

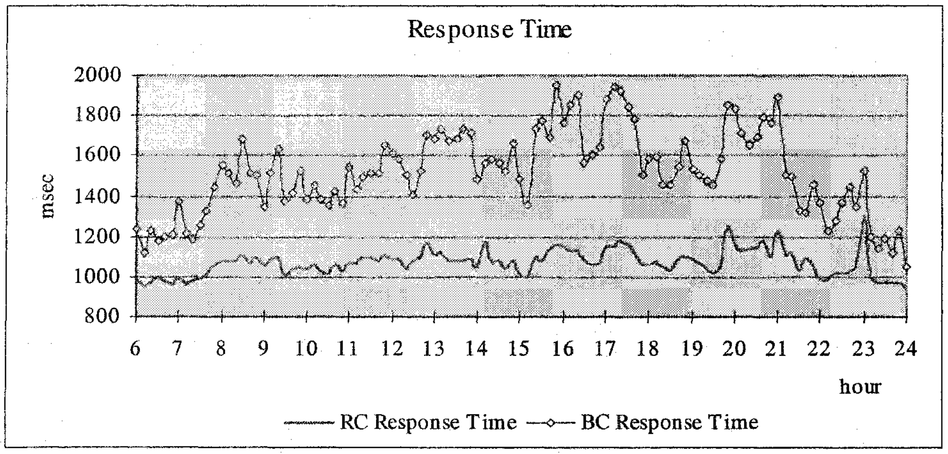


Figure 4.5 RC, BC Response Time
RC-latencyfactor=20ms,cache=120M;BC-latencyfactor=24ms,cache=100M

Under a higher network latency (SC Latency Factor = 40 ms, RC Latency Factor= 80ms, BC Latency Factor=96) as shown in Figures 4.6 and 4.7, the SC Acceptance Rate decreases, and the average response time for both RC and BC requests increases.

Figures 4.8 and 4.9 show results when the network latency is very large (SC Latency Factor = 60 ms, RC Latency Factor=120 ms, BC Latency Factor=144 ms, with other parameters unchanged). From Figures 4.4, 4.6, and 4.8, we can see that, with the increase of SC latency factor, the acceptance rate decreases more sharply with the increase of SC request intensity. When the request intensity is high, the SC processing time in a cache server is long, so a SC request tends to be forwarded to remote cache servers. However, due to the increase in link latency, more requests fail to meet the delay bound. So significantly fewer requests are accepted during high intensity periods.

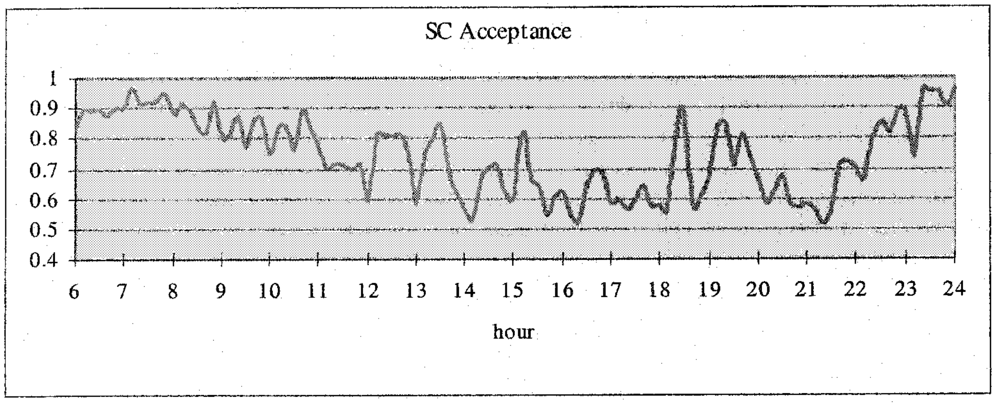


Figure 4.6 SC Acceptance Rate
DB=800ms, DBandwidth=80k,latencyfactor=40ms, cache =80M

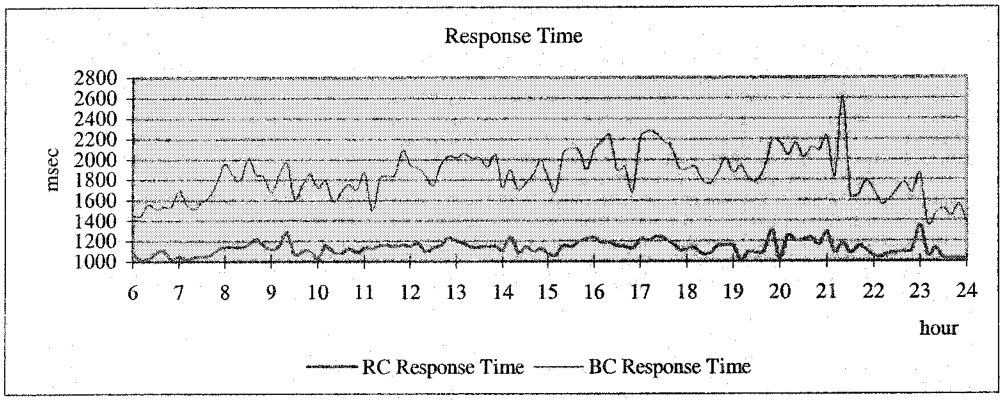


Figure 4.7 RC, BC Response Time
RC-latencyfactor=80ms,cache=120M; BC-latencyfactor=96ms,cache=100M

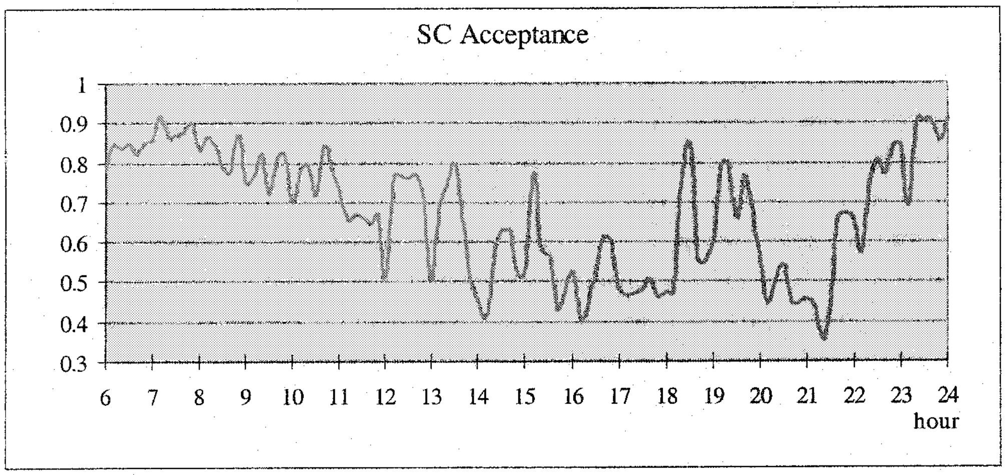


Figure 4.8 SC Acceptance Rate
DB=800ms, DBandwidth=80k, latencyfactor=60ms, cache =80M

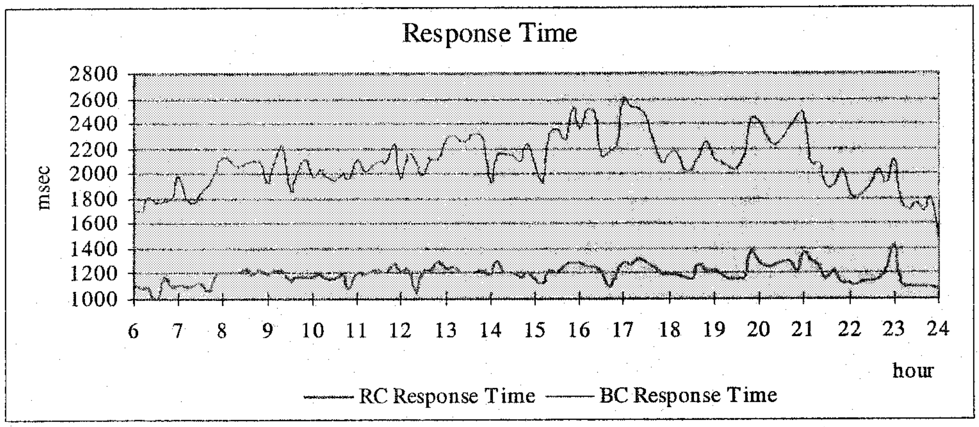


Figure 4.9 RC, BC Response Time
RC latencyfactor=120ms,cache=120M; BC latencyfactor=144ms,cache=100M

For RC and BC requests, network latency and cache workload are the two major factors influencing response time. When the network latency is low, the workload of cache servers is the main factor affecting the response time. When the workload of the cache server is light, the network latency becomes the main factor affecting the response time. From Figures 4.5, 4.7, 4.9, we see that the increase in latency factors means RC requests take a longer time. When the RC latency factor increases from

20ms to 120ms, the average RC response time increases by 6%. Similarly when the latency factor increases from 24ms to 144ms, the average RC response time increases by 24%. Since Workload Routing only considers the workload of each cache server, a request could be forwarded to a remote cache server even when network latency increases, so the average response time for BC requests increases. However, MRT always redirects the requests to the cache server with the minimum estimated response time, so the impact of latency factor is less on RC requests. The results also show that MRT algorithm guarantees that RC requests are serviced with minimum response time.

Figures 4.10 and 4.11 respectively show the acceptance rate for latency factors 10ms and 60ms at different DB values, and same values for other parameters as in Figures 4.4 and 4.8. When DB is relaxed from 800ms to 860ms, the impact on acceptance rate is greater at higher request intensities. This is because local processing time increases and disk-bandwidth becomes limited, so a SC request tends to be forwarded to remote

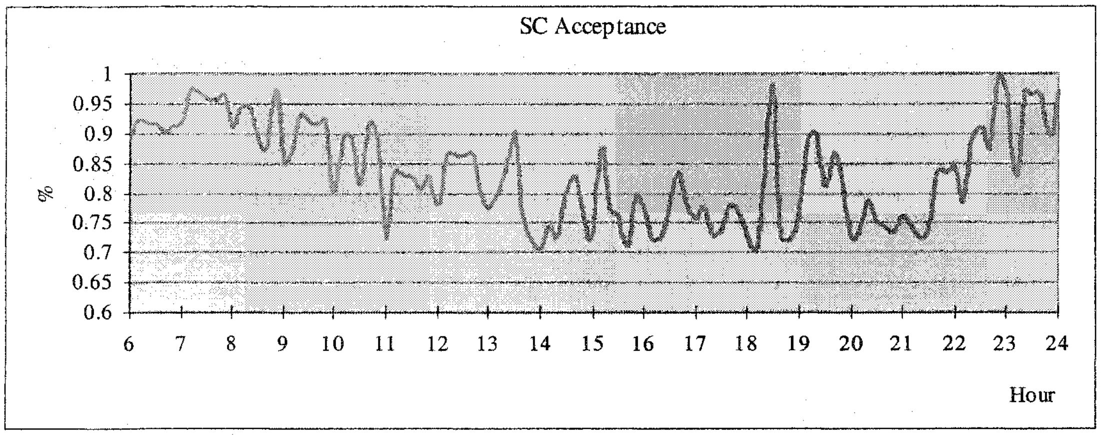


Figure 4.10 SC Acceptance Rate
DB=860ms, DBandwidth=80k,latencyfactor=10ms, cache =80M

cache servers, hence the request response time increases. The DB, which acts as a threshold, has significant impact on long-response-time requests; so more SC requests are dropped when DB is low.

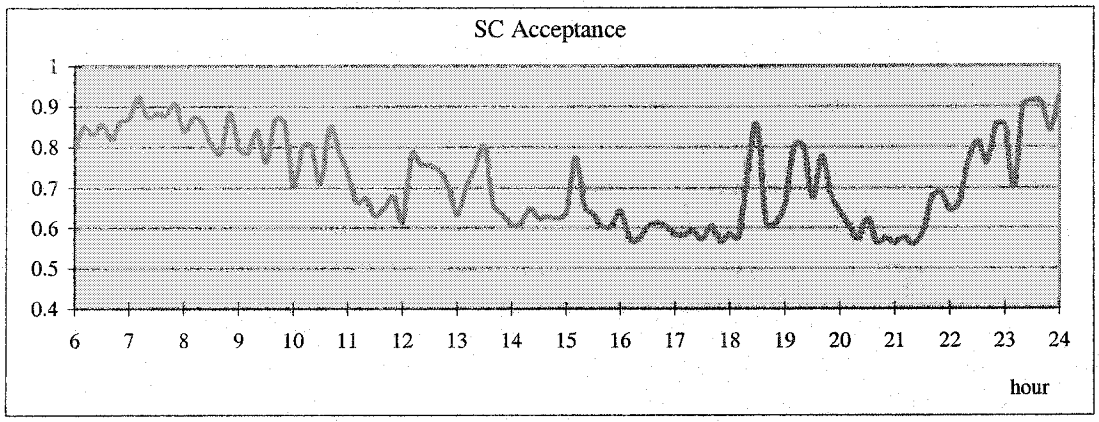


Figure 4.11 SC Acceptance Rate
DB=860ms, DBandwidth=80k, latencyfactor=60ms, cache =80M

4.4.2 Controlled-trace Simulations

We investigate the effect of the disk-bandwidth, delay bound, ratio of SC, RC and BC requests, HTTP request intensity, ratio of cache size allocation, client cluster request intensity, and the network latency factors. All simulations are run under the condition that client clusters send similar numbers of HTTP requests per 10-minute periods except when requests are explicitly unbalanced.

We study the effect of different parameters on the performance of SC, RC and BC requests by starting with a basic configuration and then varying each parameter in turn. The basic configuration is as follows: Delay bound (DB)=800ms, DBandwidth=80k/sec, SC latencyfactor=10 ms, latency factor for RC & BC=20 ms, ratio of cache allocation SC:RC:BC=8:11:11 (or 80MB:110MB:110MB), SC request intensity=200 per ten minutes, RC request intensity=900 per ten minutes, BC request intensity=900 per ten minutes. All other parameters for BC and RC are set at the same values as those in Tables 4.1-4.4. Using these settings, we obtain the following baseline results: average AR= 87%, average RC Response Time=981 ms, average BC Response Time=1263 ms.

4.4.2.1 Effect of Disk-bandwidth

We vary disk-bandwidth, and run simulations under different latency factors. Figure 4.12 shows that when disk-bandwidth increases, the number of SC requests that can be serviced increases. Obviously, when there is more disk-bandwidth, fewer requests will be dropped. The Figure also shows when disk-bandwidth is high enough (DBandwidth=160 kbytes/sec), further increasing its value hardly increases AR.

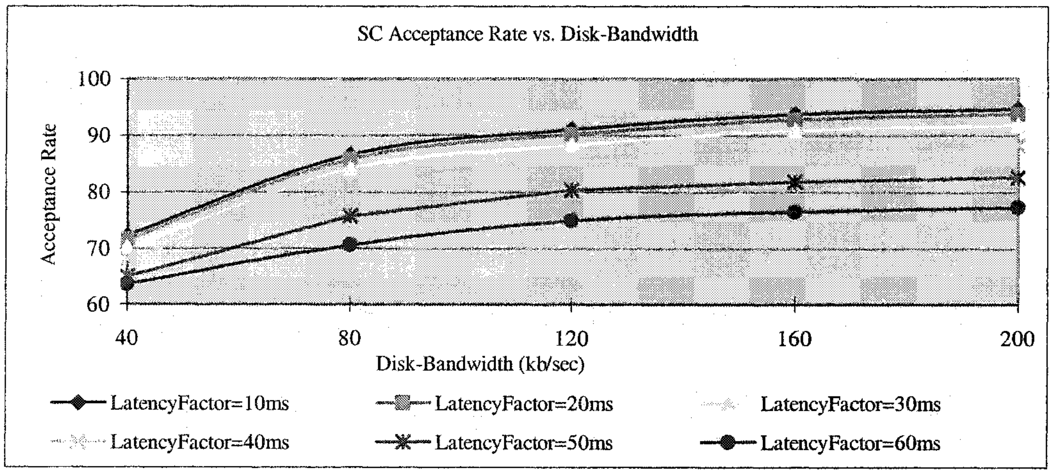


Figure 4.12 SC Acceptance Rate vs. Disk-bandwidth

4.4.2.2 Effect of Delay Bound

The effect of delay bound on SC Acceptance Rate and RC, BC response time is shown in Figures 4.13 and 4.14.

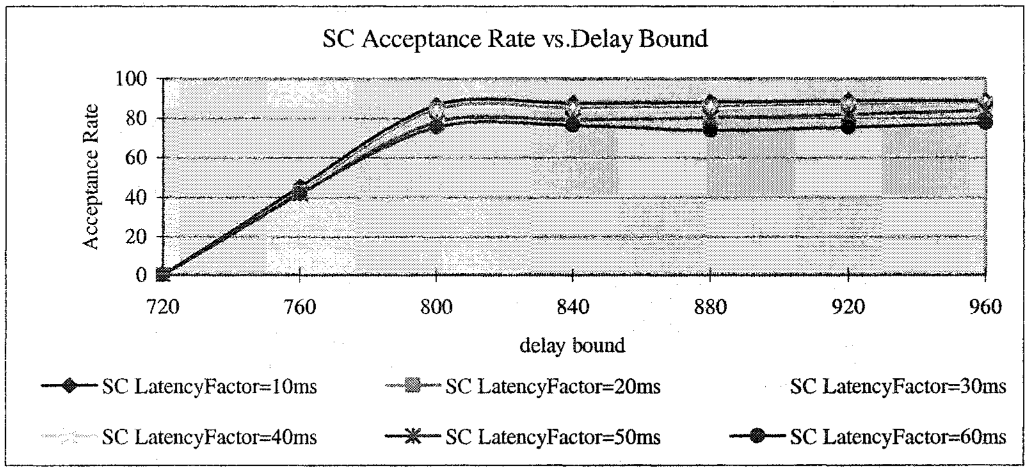
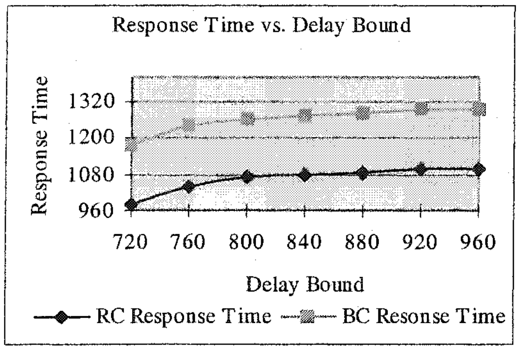


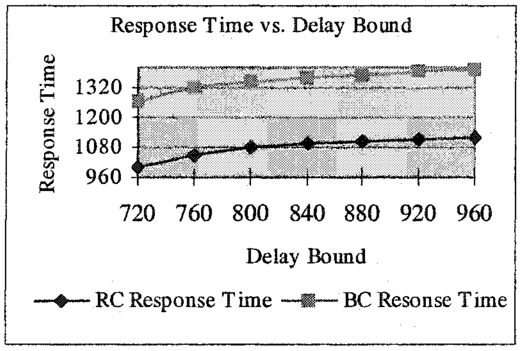
Figure 4.13 SC Acceptance vs. Delay Bound

In Figure 4.13, we see that increasing the delay bound (DB) has a positive impact on Acceptance Rate (AR) up to a certain point. DB has a more significant impact on AR for lower latencies because lower latencies result in fewer request drops. When DB increases to 920ms, AR for SC latency factor 10 ms doesn't increase, and AR for SC latency factors 20ms and 30ms changes by only 0.82% and 1.3%, respectively. This is because when DB is high enough, few SC requests are dropped because they cannot meet DB. On the other side, when DB is very low, such as 760ms, the Acceptance Rate decreases sharply; when DB reaches 720ms, no SC requests are accepted because no request can be serviced within DB.

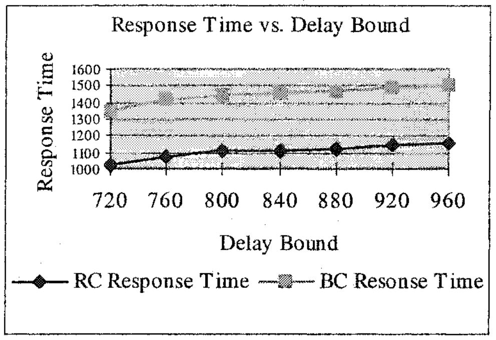
In Figure 4.14, we find that when DB increases, RC and BC response times also increase. As DB increases, more SC requests can be serviced by the system, which increases the SC workload of cache servers. Since SC workload affects RC workload, which in turn affects BC workload, it follows that an increase in DB results in an increase of RC and BC workloads, hence increasing the response time of requests from these two classes.



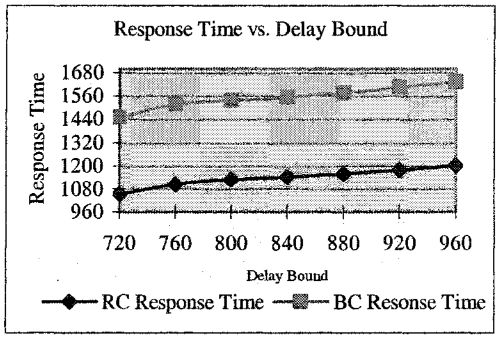
(a) Latency Factor=20ms



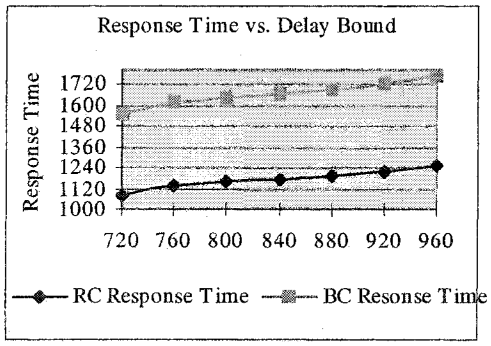
(b) Latency Factor=40ms



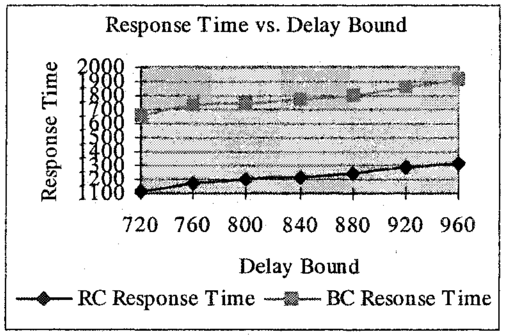
(c) Latency Factor=60 ms



(d) Latency Factor=80 ms



(e) Latency Factor =100 ms



(f) Latency Factor=120 ms

Figure 4.14 RC, BC Response Time vs. SC Delay Bound

4.4.2.3 Effect of HTTP Requests Intensity

Here, we keep the request intensity ratio SC: RC: BC=1:4.5:4.5 but vary the overall request intensity from 50%, to 250% of the baseline configuration. The results of SC acceptance rate and RC, BC response times are shown in Figures 4.15 and 4.16, respectively.

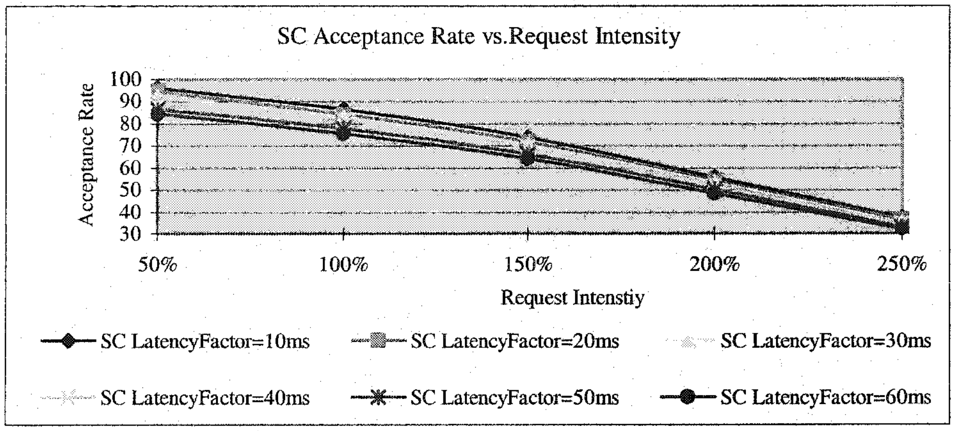
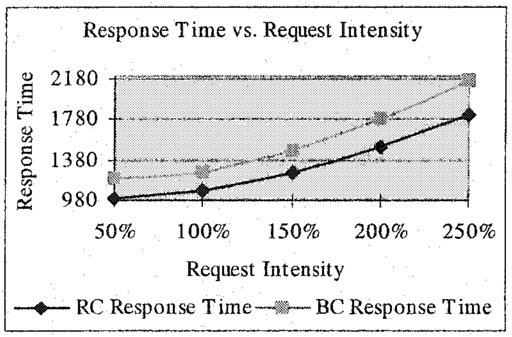


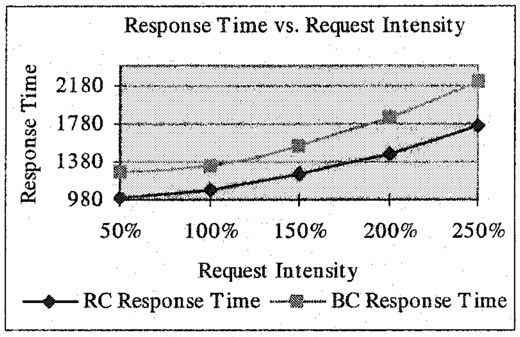
Figure 4.15 SC Acceptance Rate vs. Request Intensity

Figure 4.15 shows that when the request intensity increases, the SC Acceptance Rate decreases, and the rate of decrease is more significant. The average rate of decrease of AR is 10.1%, 14.9%, 24.2% and 33.48% when the request intensity increases from 50% to 100%, 150%, 200% and 250%, respectively. When more requests arrive during a fixed time period, the SC workload of cache servers tends to increase, so the average SC response time increases, and less SC requests are accepted. Furthermore, with the increase of request intensity, the disk-bandwidth becomes a bottleneck, so less SC requests are accepted.

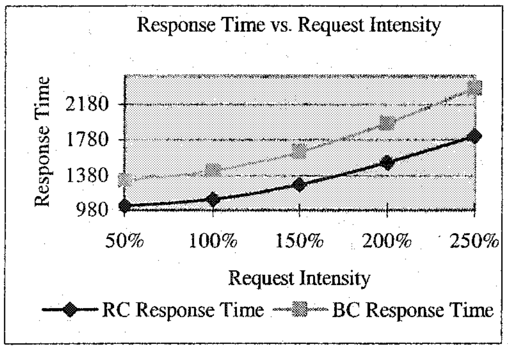
Simulation results show that the RC and BC response time is proportional to the request intensity increases (see Figure 4.16) from 50% to 250% of the baseline configuration. When RC and BC request intensity increases, more RC and BC requests are dropped due to the shortage of disk-bandwidth and due to the heavy workload at cache servers. Figure 4.16 shows that the rate of increase of RC and BC response time also increases.



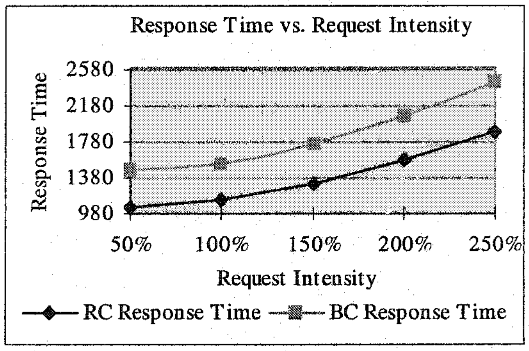
(a) Latency Factor= 20 ms



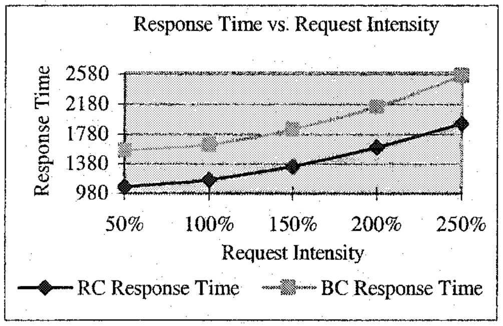
(b) Latency Factor= 40 ms



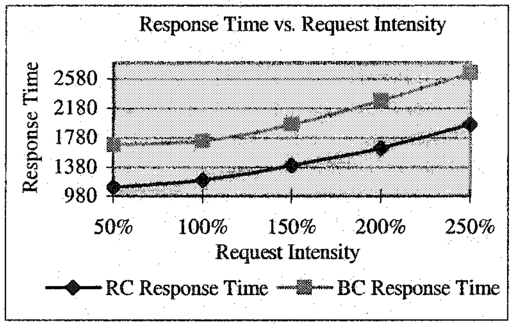
(c) Latency Factor= 60 ms



(d) Latency Factor= 80 ms



(e) Latency Factor= 100 ms



(f) Latency Factor= 120 ms

Figure 4.16 RC, BC Response Time vs. Request Intenstiy

The increase rate for RC (BC) response time is 8.0% (5.5%), 16.5% (14.4%), 18.8% (18.4%) and 20.1% (19.2%), when the request intensity increases from 50% to 100%, 150%, 200% and 250%, respectively. When request intensity increases, more requested RC and BC objects are retrieved from the original web server or remote cache servers, thus the response times increase sharply.

We also examine the SC Acceptance Rate when fixing SC request intensity, and varying the RC and BC request intensity. The results are shown in Figure 4.17.

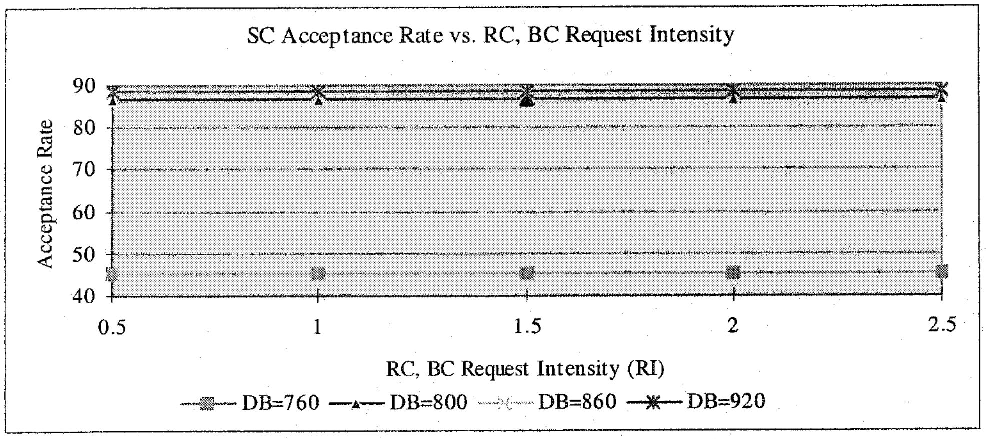


Figure 4.17 SC Acceptance Rate vs. RC, BC Request Intensity

We can see that from the Figure that no matter how we vary the request intensity of RC and BC, while fixing the request intensity of SC, the SC acceptance rate does not change. This is a very important property for our system, since SC class is of the highest priority, increasing the intensity of other requests should not influence the performance of SC class. This then satisfies our service differentiation goal.

4.4.2.4 Effect of Ratio of Cache size Allocation

We assume the total cache size is fixed for a cache server at 300M. We vary the SC cache size from 40M, 80M, 120M, 160M to 200M, and the RC and BC cache size from 130M, 110M, 90M, 70M to 50M each. Figures 4.18 and 4.19 plot the results.

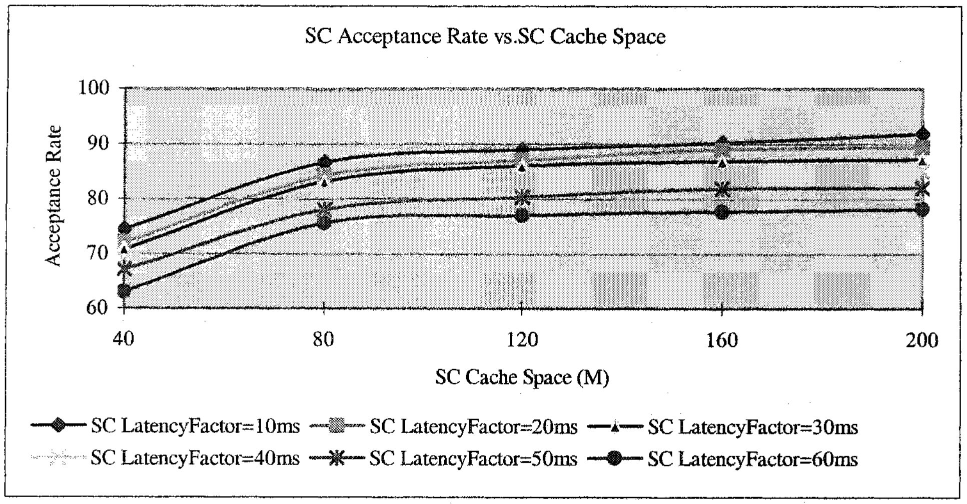
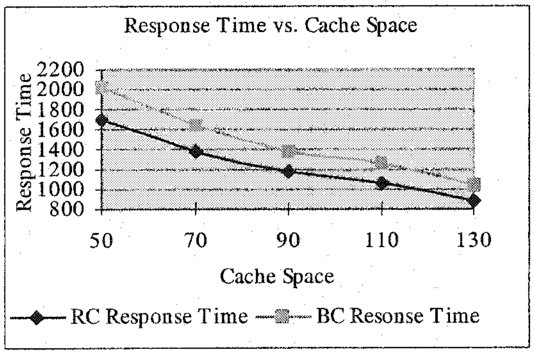


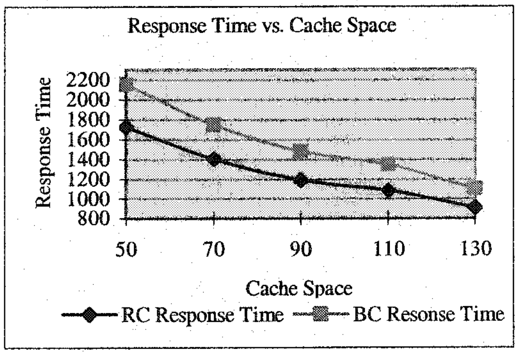
Figure 4.18 SC Acceptance Rate vs. SC Cache size

Figure 4.18 shows that when SC cache size increases, the SC Acceptance Rate also increases. With more cache size, more prefixes can be cached in a cache server, and less SC requests are dropped because the prefixes cannot be found in any cache server. In our simulations, from 40M to 80M, the increase is significant, the average increase is 17.46%, because cache size is the bottleneck for AR up to 80M. However, after 80M, the AR increases are insignificant, the rates of increase are 2.85%, 1.41% and 0.65% when SC cache space increases to 120M, 160M and 200M, respectively. This is because the factors that impede the increase of AR in this case can be disk-bandwidth, delay bound or request intensity.

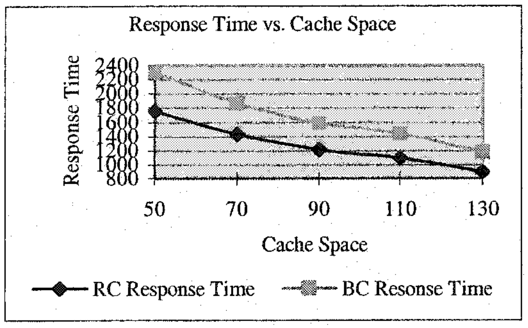
Figure 4.19 shows when RC and BC cache size decreases, the average response time for both classes increases. Less cache size means less RC and BC objects are stored in cache servers, so more RC and BC requests need to be forwarded to the original web server. When the web server has more workload, its processing time increases significantly, which results in the increase of response time of RC and BC requests.



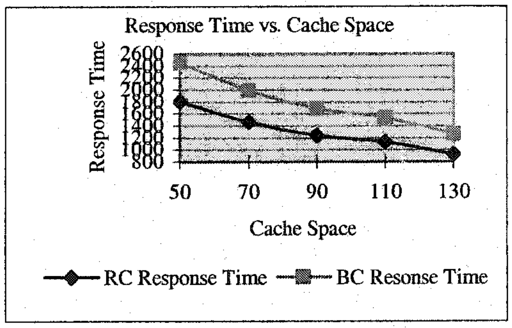
(a) Latency Factor =20 ms



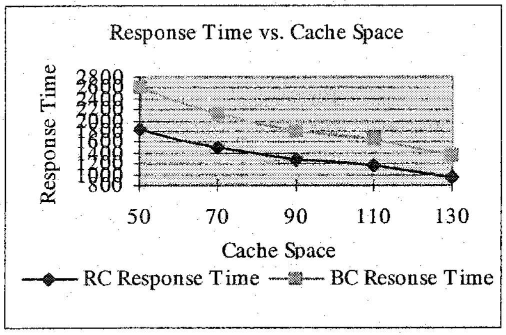
(b) Latency Factor =40 ms



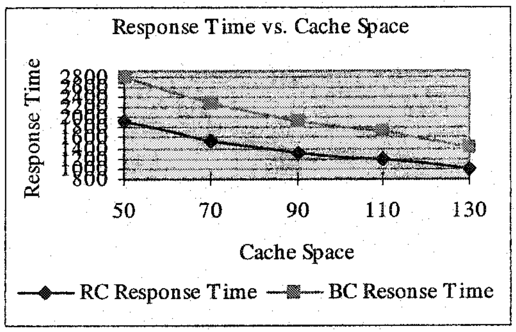
(c) Latency Factor =60 ms



(d) Latency Factor = 80 ms



(e) Latency Factor =100 ms



(f) Latency Factor =120 ms

Figure 4.19 RC, BC Response Time vs. Cache size

Beyond the value 110, the response times decrease sharply, because when the size reaches 110, a significant larger number of requests can be cached at cache servers, rather than from the web server.

4.4.2.5 Effect of Unbalanced Requests

In this set of experiments client clusters 1 and 3 produce 50% of the original HTTP request intensity and client clusters 2 and 4 produce 150% of the original HTTP request intensity. We present the results in Figures 4.20, 4.21 and 4.22, which show the effect on SC Acceptance Rate, RC and BC response time, and cache server workload, respectively.

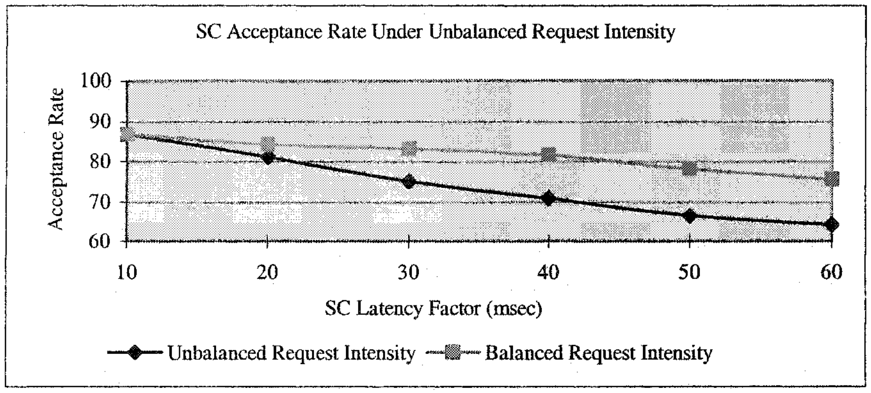


Figure 4.20 SC Acceptance Rate Under Unbalanced Request Intensity

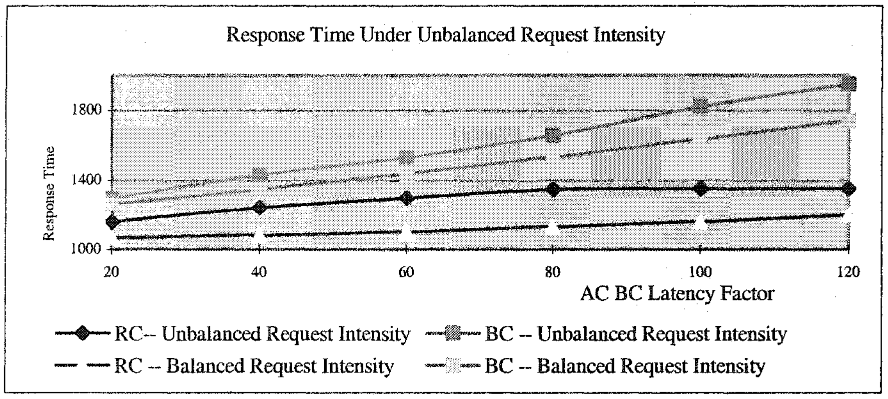


Figure 4.21 RC, BC Response Time Under Unbalanced Request Intensity

We see from Figure 4.20 that an unbalanced request intensity causes AR to be less than that of balanced request intensity. With unbalanced request intensity, more SC requests tend to be forwarded to remote cache servers and the heavily loaded cache servers take longer time to process requests, so the response time increases.

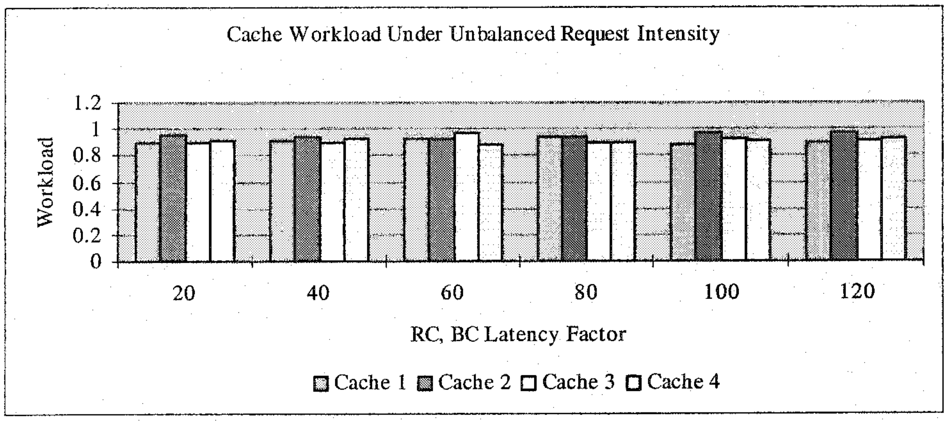


Figure 4.22 Cache Server Workload Under Unbalanced Request Intensity

We also see in Figure 4.21 that an unbalanced workload causes RC and BC response times to increase. For RC, heavily loaded cache servers take longer to process a request; so more requests are forwarded to remote cache servers. The overall average response time is therefore increased. For BC, a switch whose cache server is heavily loaded forwards more requests to remote cache servers with less workload, so its average response time increases as well. Workload routing for BC requests, however, moves the extra load from the heavily loaded cache servers to the lightly loaded cache servers so that the workload on all cache servers is balanced, as shown in Figure 4.22. When the workload is balanced, the average processing time in cache servers is reduced.

4.4.2.6 Effect of Unbalanced Cache Size

The cache sizes of the four cache servers are different in this experiment. We fix the overall cache size to be 1200M, assign 50% of basic cache size, or 150M to cache server 1 and 3 each, and assign 150% of basic cache size, or 450M to cache server 2 and 4 each. Within each cache server, the ratio of cache size allocated to SC to that allocated to RC to that allocated to BC is the same for all four cache servers, which is 8:11:11, the same as the basic configuration. Thus, in cache servers 1 and 3, 40M is allocated to SC, and 55M is allocated to each of RC and BC; in cache servers 2 and 4, 120M is allocated to SC, and 165M is allocated to each of RC and BC. We also let client clusters 1 and 3 have 50% of basic HTTP request intensity, and client clusters 2 and 4 have 150% of basic HTTP request intensity. Figures 4.23, 4.24 and 4.25, respectively show the acceptance rate of SC, response times of RC and BC, and cache server workload.

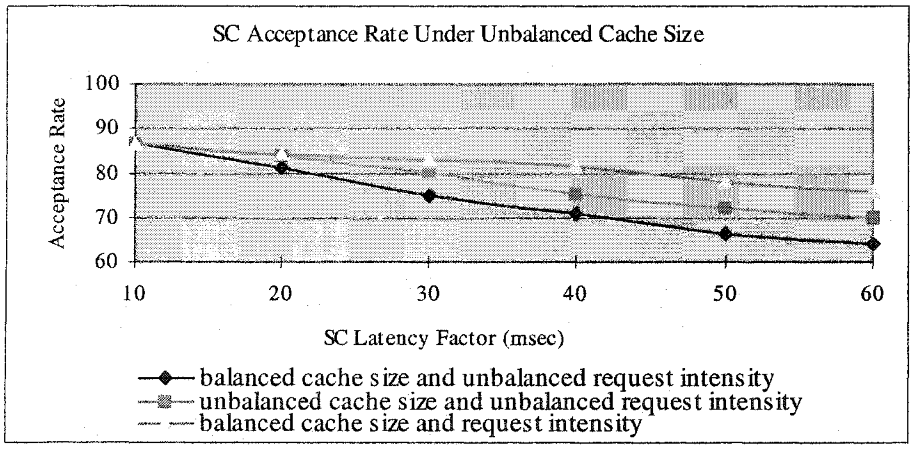


Figure 4.23 SC Acceptance Rate Under Unbalanced Cache Size and Unbalanced Request Intensity

Figure 4.23 shows the performance of the unbalanced cache size and unbalanced request intensity case outperforms the case of unbalanced request intensity, but balanced cache size. Compared with balanced cache size with unbalanced request intensity, unbalanced cache size with unbalanced request intensity produces most requests in client clusters closer to the larger cache servers, thus more requests can be served locally instead of being forwarded to remote cache servers, so less requests tend to be dropped because the estimated response time is greater than the delay bound. Compared with balanced cache size and balanced request intensity, although a large cache server could cache more objects, the limited disk-bandwidth allows only a limited number of requests to be handled simultaneously, so a large number of requests forwarded to a cache server lead to less acceptance rate.

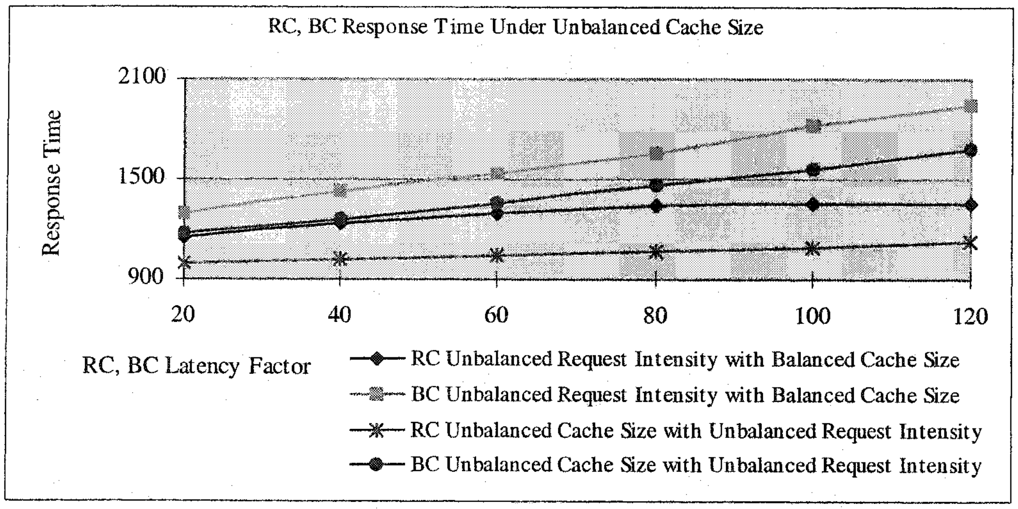


Figure 4.24 RC, BC Response Time Under Unbalanced Cache Size and Unbalanced Request Intensity

From Figure 4.24, we see that the response time for RC and that for BC under unbalanced cache size with unbalanced request intensity is worse than that for RC and that for BC under balanced cache size with unbalanced request intensity. The reason is that more RC and BC requests come to larger local cache servers, and so more requests are serviced locally instead of remotely.

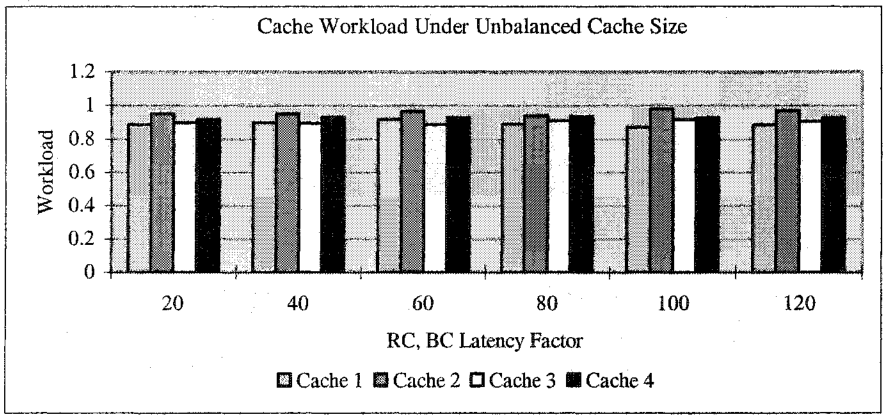


Figure 4.25 Cache Server Workload Under Unbalanced Cache Size

From Figure 4.25, we see that Workload routing for BC traffic can still balance the workload on the different cache servers, even if more SC and RC requests are directed to the larger cache servers.

4.5 Summary

In this chapter we studied the performance of the proposed SDSC, which evaluates the AR of SC, compares the response times for RC and BC. The system model, the network latency model, the workload model and the simulation parameter settings have been described. Implementation methods of DBR, MRT and web server are presented. Two types of simulation experiment - raw trace and controlled-trace simulation are presented to show the effects of the disk-bandwidth, delay bound, HTTP request intensity, cache size, network latency factors and unbalanced requests on the performance of SDSC.

The simulator we developed to observe and evaluate the performance of SDSC is trace-driven. We used the original NLANR log traces as input; also we modified the traces to obtain different request intensity and request distribution among the three classes, and observed performance of SDSC under these circumstances. Generally, the acceptance rate of SC increases when disk-bandwidth increases, delay bound is relaxed, request intensity decreases, or more cache space is allocated to SC objects. RC requests always have lower response times than BC requests given that all parameter setting for the two classes are the same. The workload of cache servers is well balanced under any circumstances. The results satisfy our system requirement that RC requests have a higher priority than BC requests in terms of response time in the system.

Chapter 5 Conclusion

The objective of our research is to study how differentiated service can be facilitated with a switching-based web caching system. We proposed the deployment of Switching-based Differentiated Service Caching (SDSC) that classifies incoming requests into three classes - streaming class (SC), real-time assured class (RC), and best-effort class (BC), based on the type of HTTP requests. SDSC uses a different routing algorithm for each class, namely Disk-Bandwidth Routing (DBR), minimum response time (MRT), and Workload Routing (WR) for the SC, RC and BC classes, respectively. DBR guarantees that accepted SC requests are served with minimum response time. However, only requests that are expected to be serviced within a specified delay bound (DB), and for which there is a prefix in a cache with enough disk-bandwidth, can be accepted.

Our simulations show that no matter how request intensity patterns change for RC and BC, they do not influence the acceptance rate (AR) of SC, which achieves higher priority for SC requests. AR of SC depends on the disk-bandwidth consumption of cache servers, delay bound of SC, network latency, the request intensity of SC, SC workload at cache servers, and so on. Adjusting any of these factors within a certain limit while fixing others influences AR, beyond this limit the adjustment has minimal impact or no impact on AR.

We also compared the performance of RC and BC. RC requests are redirected to the cache server with the minimum estimated HTTP request response time by MRT, which estimates the time based on cache server content, cache server workload, the web server workload and network latency. BC requests are redirected to the cache server with the minimum workload; hence balancing the workload of cache severs, which in turn can improve the performance of SC and RC. The response time of RC requests is shorter than that of BC given the same parameter settings for both classes. Because of WR, the workload of cache servers is well balanced under all conditions.

5.1 Contributions

The contributions of the research are summarized as follows:

1. We studied service differentiation in web caching systems, and proposed SDSC to classify HTTP requests and achieve DiffServ for the classified requests. SDSC considers disk-bandwidth, delay bound, cache server content, cache server workload and link delay to make acceptance and routing decisions. SDSC guarantees SC requests to be serviced with minimum response time and within a delay bound, RC requests with the minimum possible response time. SDSC balances the workload of all cache servers.
2. A comprehensive simulation was developed to evaluate the performance of SDSC. We evaluated how factors such as delay bound, disk-bandwidth, and latency factor influence the acceptance rate of SC requests. We compared the performance of RC requests with that of BC requests. Simulation results show that RC requests

always have lower response times than BC requests, SC is of higher priority than RC and BC, and workload of cache servers is always balanced.

5.2 Future Work

The following aspects of our research need further investigation:

1. DiffServ at Transport layer: In this thesis, we propose to achieve DiffServ at the Application layer, which utilizes DiffServ at Network layer and Transport layer. DiffServ at the Network layer has already been proposed and can be used directly in SDSC, but DiffServ at the Transport layer in SDSC needs further study.
2. Cache Server DiffServ: Further investigation is needed to achieve DiffServ at cache servers, in terms of serving different classes with different processing rates and priorities.
3. Dynamic Content: Dynamic content is one of the classes of content on the Web. Dynamic content is different from the content we have classified in the research in that it is user-driven, so if a dynamic page is retrieved from the originating server and stored in a cache, when a user retrieves it from the cache later, the fetched page may not desirable anymore, since the information is no longer current, or it does not satisfy the tailored requests of customers [9].
4. Extending DiffServ Metrics: To better achieve DiffServ, we need consider more DiffServ metrics such as jitter for streaming class, drop rate of assured class and best-effort class [19], etc.

References

[1] Tarek F. Abdelzaher, Kang G. Shin, Nina Bhatti, "Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach" http://computer.org/tpds/td2002/l0080abs.htm, 2002

[2] J. Almeida and P.Cao, "Measuring Proxy Performance with the Wisconsin Prxoy Benchmark". In *Proceedings of the Third International Caching Workshop*, June 1998

[3] Jussara Almeida, Mihaela Dabu, Anand Manikutty and Pei Cao, University of Wisconsin-Madison, "Providing Differentiated Levels of Service in Web Content Hosting", http://www.cs.wisc.edu/~cao/papers/diff-QoS.html

[4] Michael J. Andrews and David Cyganski, Convergent Technology Center, WPI Worcester, MA, "Characteristics of Multimedia Data", http://www.gweep.net/~rocko/XUDP_Paper/node2.html

[5] Apache HTTP Server Project, http://httpd.apache.org/

[6] G. Apostolopoulos, V. Peris, P. Pradhan, , IBM Research Division, "IBM Research Report L5: A Self Learning Layer-5 Switch", http://citeseer.nj.nec.com/cache/papers/cs/14107/http:zSzzSzwww.research.ibm.comzSzpeoplezSzdzSzdebanjanzSzpaperszSzl5.pdf/l-a-self-learning.pdf

[7] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", In *Proceedings of the 1998 RCM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, July 1998

[8] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz, Bell Laboratories, Lucent Technologies, "The Eclipse Operating System: Providing Quality of Service via Reservation Domains", *http://www.cs.washington.edu/sosp16/bruno.html*

[9] Pei Cao, Jin Zhang and Kevin Beach Computer Sciences Department, University of Wisconsin-Madison, "Active Cache: Caching Dynamic Contents on the Web", http://www.cs.wisc.edu/~cao/papers/active-cache.html, 1998

[10] Surendar Chandra, Carla Schlatter Ellis and Amin Vahdat, "Differentiated Multimedia Web Services Using Quality Aware Transcoding", *INFOCOM 2000 - Nineteenth Annual Joint Conference of the IEEE Computer And Communications Societies*, Mar 2000

[11] Wu-chang, Feng, Puma Technology, "Building Scalable Internet Services", http://www.thefengs.com/wuchang/work/cs444i.ppt

[12] ArrowPoint Communications, "Content Smart Cache Switching", White paper.

[13] Brian D. Davison, "Web Caching", http://www.web-caching.com/mnot_tutorial/intro.html#PROXY, 2001

[14] Cisco "CSS-11150 Content Services Switch", http://www.cisco.com/univercd/cc/td/doc/pcat/11150.htm

[15] Cisco, "Quality of Service Networking", http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/qos.htm#xtocid1

[16] Distributed Web Caching project. Available at: http://ringer.cs.utsa.edu/research/proxy/proxy.html

[17] Pei Cao, "Bloom Filters - the math", http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html, July, 1998

[18] C. Faloutsos and S. Christodoulakis, "Design of a Signature File Method that Accounts for Non-Uniform Occurrence and Query Frequencies". In Proceedings of *11th International Conference on VLDB*, pp. 165-170, Stockholm, Sweden, August 1985

[19] Mark Fishburn, "Benchmarking Differentiated Service", http://www.ietf.org/proceedings/00dec/slides/BMWG-2/, August 2001

[20] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance". *ITTT/RCM Transactions on Networking*, August 1993

[21] HCL Technologies Ltd., "Streaming Media", http://cdn.hcltech.com/StreamingMedia.htm

[22] IETF, "An Expedited Forwarding PHB", http://www.ietf.org/rfc/rfc2598.txt, 1999
[23] IETF, "An Informal Management Model for Diffserv Routers", http://www.ietf.org/rfc/rfc3290.txt, 2002
[24] IETF, "Assured Forwarding PHB Group", http://www.ietf.org/rfc/rfc2597, 1999
[25] IETF Network Working Group, "An Architecture for Differentiated Services", http://www.faqs.org/rfcs/rfc2475.html, December 1998
[26] IETF Network Working Group, "Differentiated Services", http://www.ietf.org/proceedings/98dec/43rd-ietf-98dec-121.html, November, 1998
[27] IETF Networking Group, "Internet Control Message Protocol", http://www.faqs.org/rfcs/rfc792.html
[28] "IRCache FAQ and Users Guide", http://www.ircache.net/FAQ/#ss1
[29] Chung-Ta King , National Tsing-Hua University, "Proxy Prefetch and Prefix Caching", http://www.computer.org/proceedings/icpp/1257/12570095abs.htm, 2001
[30] Zhengang Liang, Hossam Hassanein, Patrick Martin, *"Transparent Distributed Web Caching"*, *Proceedings of the IEEE Local Computer Network Conference*, pp225-233, Nov 2001
[31] Y. Lu, A. Saxena and T. F. Abdelzaher, University of Virginia, "Differentiated Caching Service; A Control-Theoretical Approach", In the 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, April 2001.
[32] Theresa-Marie Rhyne, RCM SIGGRAPH Carto Project Director, "Exploring the Concept of Streaming Media for Geographic Visualization", http://www.siggraph.org/~rhyne/carto/ica-3.htm
[33] M. Rabinovich, "Not All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network". In *Computer Networks And ISDN Systems*, 30, 22-23, pp.2253-2259, Nov 1998
[34] M. Ripeanu, A. Iamnitchi, "Bloom Filters – Short Tutorial", http://www.flipcode.com/tutorials/tut_bloomfilter.shtml
[35] P. Rodriguez, C. Spanner, and E. biersack, "Web caching architectures: Hierarchical and distributed caching". In *Proceedings of the 4th International Web Caching Workshop,* April 1999.
[36] A. Rousskov and V.Soloviev, "On performance of caching Proxies". In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98/PERFORMANCE '98)*, pp. 272-273, Madison, W1, June 1998
[37] A. Rousskov and D. Wessels, "Cache digests", *Proceedings of the Third International WWW Caching Workshop*, Manchester, England, June 1998.
[38] M.Sayal, Y.Breitbart, P.Scheuermann, and R. Vingralek, "Selection algorithms for replicated web servers". In *Performance Evaluation Review*, vol.26, no.3, pp.44-50, Dec.1998.
[39] S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams", http://citeseer.nj.nec.com/cache/papers/cs/3549/ftp:zSzzSzgaia.cs.umass.eduzSzpubzSzsenzSzSen_ProxyPrefix_Infocom99.pdf/sen99proxy.pdf, 1999
[40] R.Tewari, H.Vin, A.Dan, and D.Sitaram, "Resource-Based Caching for Web Servers", in Proc. Of SPIE/ACM Conference on Multimedia Computing and Networking, San Jose, USA, Jan. 1998, pp.191-204
[41] X. Tang, F. Zhang, S. T. Chanson , Hong Kong University of Science and Technology, "Streaming Media Caching Algorithms for Transcoding Proxies", in 2002 International Conference on Parallel Processing (ICPP'02) http://www.computer.org/proceedings/icpp/1677/16770287abs.htm, August 2002
[42] D. Wessels and K. Claffy, "Internet Cache Protocol (ICP), version2". RFC 2186, Sep 1997
[43] Haoming Wu, "Least Relative Benefit Algorithm for Caching Continuous Media Data at the Web Proxy", master thesis, 2001
[44] Y. Zhou and J. F. Philbin, NEC Research Institute, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches" http://www.usenix.org/events/usenix01/full_papers/zhou/zhou_html/index.html
[45] Qing Zou, Patrick Martin, Hossam Hassanein, "Transparent Web Caching with Minimum Response Time", *Proc. Of IEEE IPCCC*, April 2003

Appendix A Bloom Filter

The difference between a weighted Bloom Filter and a Bloom Filter is that in a weighted Bloom Filter the objects with high frequency are represented by more bits while the objects with low frequency are represented by fewer bits. In a regular Bloom Filter, all objects in a cache server are represented by the same number of bits.

In a weighted Bloom Filter, we assume that, according to some conditions such as frequency the set S of all objects in a cache can be partitioned into n subsets S_1, $S_2 \ldots S_n$, which are disjoint and whose union is S, that is,

$$S = S_1 \cup S_2 \ldots \cup S_n$$

And

$$S_i \cap S_j = \varnothing, \text{ where } 1 \le i \le n,\ 1 \le j \le n, \text{ and } i \ne j$$

We define the following variables:

D_i: The number of objects in subset S_i, $D = D_1 + D_2 + \ldots + Dn$ is the total number of objects in the cache.

P_i: The access probability for objects in S_i. It is the possibility that any object in subset S_i will be accessed.

W_i: The weight for subset S_i. It is the number of hash functions for subset S_i

F: The Bloom Filter size

In a weighted Bloom Filter representing D objects, the probability that a particular bit is 0 is:

$$R_0 = (\frac{F-1}{F})^{W1*D1 \ + W2*D2 \ +...+ Wn*Dn} = (1-\frac{1}{F})^{W1*D1 + W2*D2+...+ Wn*Dn} \quad \text{(B.1)}$$

We know that $(1 - \frac{1}{x})^{b} = e^{-b/x}$ when x -> ∝

Equation (1) can be approximated as:

$$R_0 \approx e^{-(W1*D1 + W2*D2+...W3*D3)/F}, \quad \text{when F -> ∝} \quad \text{(A.2)}$$

Hence the probability that a particular bit is 1 is:

$$R_1 = 1 - R_0 \approx 1 - e^{-(W1*D1 + W2*D2+...Wn*Dn)/F} \quad \text{(A.3)}$$

The false prediction probability is:

$$Fp = P_1*R_1^{W1} + P_2*R_1^{W2} +...+Pn*R1^{Wn} \quad \text{(A.4)}$$

To find the optimum W_i for each subset S_i such that the false prediction F_p is minimized, we differentiate F_p with respect to W_i:

$$\frac{\partial F_p}{\partial W_i} = 0, \quad \text{where } 1 \le i \le n$$

$$\frac{\partial F_p}{\partial W_i} = \frac{R}{1-R}\frac{F}{D_i}P_i R^{W_i}\ln R + \sum_{j=1}^{n} P_j W_j R^{W_j} = 0, \quad 1 \le i \le n \quad \text{(A.5)}$$

Equation above is equivalent to:

$$\frac{P_1 R^{W_1}}{D_1} = \frac{P_1 R^{W_1}}{D_1} = ... = \frac{P_1 R^{W_1}}{D_1} = \frac{F_p}{D} = K \quad \text{(A.6)}$$

K is a constant independent of i.

Substituting equation (A.6) into (A.5), then

$$K\left[F\frac{R}{1-R}\ln R + \sum_{j=1}^{n} W_j D_j\right] = 0$$

or

$$\frac{R}{1-R} = \frac{\sum_{j=1}^{n} W_j D_j}{F \ln R} \tag{A.7}$$

From equations (A.6) and (A.7),

$$\frac{R}{1-R} = \frac{\ln(1-R)}{\ln R} \text{ or } R = \frac{1}{2} \tag{A.8}$$

Substituting equation (A.8) into (A.7), (A.8)

$$\sum_{j=1}^{n} W_j D_j = F \ln 2 \tag{A.9}$$

Substituting equation (A.8) to (A.6),

$$W_i = \frac{1}{\ln 2}\left[\ln \frac{p_i}{D_i} - \ln K\right] \tag{A.10}$$

Substituting equation (A.10) to (A.9),

$$\ln K = \frac{-F(\ln 2)^2 + \sum_{i=1}^{n} D_i \ln \frac{P_i}{D_i}}{D} \tag{A.11}$$

Substituting equation (A.11) to (A.10), the optimum values for W$_i$ is:

$$W_i = \frac{F \ln 2}{D} + \frac{1}{\ln 2}\left[\ln \frac{P_i}{D_i} - \frac{\sum_{j=1}^{n} D_j \ln \frac{P_i}{D_i}}{D}\right], 1 \le i \le n \tag{A.12}$$

Appendix B Simulator Structure

The simulator simulates ICP, Cache Digest, Layer 5 switch, and SDSC. It is object-oriented, event-driven, developed in C++ and runs on Linux system. It mainly consists of five types of objects, namely Client Cluster, L5 Switch, Cache Server and Web Server, each of which has an event handler to process incoming events and generate events until all requests have been processed or the time we set is up. All generated events are sent to and queued at the object EventManager, which schedules to handle events based on time ordering.

The main classes involve ClientCluster, Switch, CacheServer, WebServer, Link, EventManager, SimuObject, SimuEvent, LRUCache. Other classes such as MD5, SimuStat, SimuParam, LogEntry, LinkReg, LinkedList, LinkedListEnumerator, ICPEntry are supporting classes. The functions of the main classes are described as follows:

1. ClientCluster. It generates requests by reading log entries in log traces. It sends requests to its local link connected to local switch. When a requested object is returned or dropped, it triggers SimuStat to record related information.

2. Switch. It maintains information of cache servers such as content, workload, disk-bandwidth unused in its table called CacheInfo. It also measures and saves information such as link delay. In this class, function requestClassification categorizes incoming requests into one of the three classes, namely SC, RC and BC; DBURouting handles SC requests, MRTRouting handles RC requests, and

WorkloadRouting handles BC classes. It also schedules the web server to return the remainder of a SC object after the prefix is sent by a cache server.

3. CacheServer. It periodically informs switches of its content, SC, RC and BC workload. Objects are cached in its LRUCache. It receives forwarded requests from links connected to switches, and returns a cached object or a prefix that satisfies the request. It asks the web server for the object or the prefix if the latter is not cached and if there is disk-bandwidth requirement is met.

4. WebServer. It receives events from links connected to cache severs or switches, and generates corresponding events. It sends the remainder of a SC object to the switch. It also sends the SC prefix or RC or BC objects to cache servers.

5. Link. It receives an event from a connected object, and generates a new event to be handled by the other connected object.

6. EventManger. It uses LinkedList to queue all events generated by all objects, and dispatches events according to time order.

7. SimuObject. It defines a virtual function *eventHandle*, which is inherited by all inherited classes such as WebServer, CacheServer.

8. SimuEvent. It contains information of events, such as generator, handler, timeArrival, and so on.

9. LRUCache. It contains a cache database, and functions to operate on the database such as addToCache, discard.

Appendix C Pseudo Code

/*Pseudo code for switch*/

```
/*A switch classifies incoming requests into classes, makes routing decision. It
 *receives TCP messages, HTTP messages, ICP messages. It queries workload and
 *disk bandwidth usage periodically.*/
/* On receipt of TCP messages*/
Procedure onReceiveTCPMessages (msg:TCP, se:SimulationEvent){
  if (msg.type==TCP_SYN)          //from client
    sendTCP_ACK (clientAddress);

  if (msg.type==TCP_ACK)          //from CS or web server  {
     SimulationEvent newse;
     if (se.from=WS)
       newse. ObjSize=EntrySize-SizeOfPrefix; //only the left part needed from WS
     else //from CS
       newse. ObjSize=SizeOfPrefix;
     sendHTTP_REQ (WsorCacheAddress, newse);
  }

if (msg.type==TCP_FIN)          //from CS or WS
   if (se.objType==SC) {
     if (se.from==CS)  {}
     if (se.from==WS)  sendTCP_FIN (clientAddress);
   }
   else if (se.objType==RC || se.objType==BC)   sendTCP_FIN(clientAddress);
}

/*onReceipt of HTTP messages*/
Procedure onReceiveHTTPMessage (msg:HTTP, se:SimulationEvent){
 //HTTP_REQ only from client. Switch classifies requests into SC, RC or BC.
 //Then DBR, MRT or Workload algorithm is applied to forward the request.
 se.objType:=RequestClassification(se);
 if (msg.type==HTTP_REQ)  {
   if (se.objType==SC)  DBURouting(se);
   if (se.objType==RC)  MRTRouting(se);
   if (se.objType==BC)  WorkloadRouting(se);
 }

 if (msg.type == HTTP_RES) //it could be from CS or SW   {
   if (se.objType==SC)
    if (se.from==CS)
     if (gotPrefix==false)  {
       sendHTTP_RES (clientAddress, null);
      sendTCP_FIN (clientAddress);
     }
      else // (gotPrefix==true)  {
       sendHTTP_RES (clientAddress, prefix);
       if (objSize>prefix)
         sendTCP_SYN (WSAddress);
       else // objSize<=prefix
         sendTCP_FIN (clientAddress);
     }
    else // se.from==WS   {}
  else // (se.objType==RC|| se.objType==BC ) sendHTTP_RES (clientAddress, obj);
 }

/*on receipt of ICP messages*/
// ICP messages are those for content update,
//workload and disk bandwidth update
```

```
Procedure onReceiveICPMessage (msg: ICPMsg) {
 if (msg.OPCode==ICP_UPDATE_CONTENT)
   for i:=1 to NumOfCSs
    if (CacheArrayTable[i].SenderAddress==msg.SenderAddress) &&
     (msg.TS>CacheArrayTable[i].Last_Content_UpdateMsg_TS)  {
      CacheArrayTable[i].Content:=msg.Content;
      CacheArrayTable[i].Count:=msg.Num;
      CacheArrayTable[i].Last_Content_UpdateMsg_TS:=msg.TS;
      sendMsg (ICP_UPDATE_CONTENT_RCK, mySwitch.IPAddress,
      msg.TS, CacheArrayTable[i].SenderAddress)
    }

 if (msg.OPCode==ICP_UPDATE_WORKLOAD) //from WS
  WS.Workload=msg.Workload;

 if (msg.OPCode==ICP_UPDATE_WORKLOAD_DBANDWIDTH)
  for i:=1 to NumOfCSs
   if CacheArrayTable[i].SenderAddress==msg.SenderAddress) && (msg.TS>
       CacheArrayTable[i].Last_Workload_DBandwidth_UpdateMsg_TS)
    {
      CacheArrayTable[i]. WorkloadOfSC:=msg.Workload;
      CacheArrayTable[i]. DBandwidth:=msg.Dbandwidth;
      CacheArrayTable[i]. Last_Workload_DBandwidth_UpdateMsg_TS:=msg.TS;
      CacheArrayTable[i].Workload_DBandwidth_QueryRes_Time:=getCurrentTime();

      //roundtrip time is calculated
      if (msg.TS==CacheArrayTable[i].Workload_Query_TS)
       CacheArrayTable[i].netwokLatencyOfSC:=
       CacheArrayTable[i]. Workload_DBandwidth_QueryRes_Time-
       CacheArrayTable[i]. Workload_Query_Time;
    }

  if (msg.OPCode==ICP_UPDATE_WORKLOAD_RC)
   if (msg.SenderAddress==WS.IPAddress)  WS.Workload=msg.Workload;
   else
   {
     for i:=1 to NumOfCSs
      if CacheArrayTable[i].SenderAddress==msg.SenderAddress) &&
      (msg.TS>CacheArrayTable[i].Last_Workload_RC_UpdateMsg_TS) {
       CacheArrayTable[i].WorkloadOfRC:=msg.Workload;
       CacheArrayTable[i]. Last_Workload_RC_UpdateMsg_TS:=msg.TS;
       CacheArrayTable[i].Workload_RC_QueryRes_Time:=getCurrentTime();

       //roundtrip time is calculated
       if (msg.TS==CacheArrayTable[i].Workload_Query_TS)
         CacheArrayTable[i].netwokLatencyOfRC:=
         CacheArrayTable[i]. Workload_RC_QueryRes_Time-
         CacheArrayTable[i]. Workload_Query_Time;
      }
   }

  if (msg.OPCode==ICP_UPDATE_WORKLOAD_BC)
   if (msg.SenderAddress==WS.IPAddress)  WS.Workload=msg.Workload;
   else
   {
     for i:=1 to NumOfCSs
      if CacheArrayTable[i].SenderAddress==msg.SenderAddress) &&
      (msg.TS>CacheArrayTable[i].Last_Workload_BC_UpdateMsg_TS) {
       CacheArrayTable[i].WorkloadOfBC:=msg.Workload;
       CacheArrayTable[i]. Last_Workload_BC_UpdateMsg_TS:=msg.TS;
       CacheArrayTable[i].Workload_BC_QueryRes_Time:=getCurrentTime();
```

```
            //roundtrip time is calculated
            if (msg.TS==CacheArrayTable[i].Workload_Query_TS)
              CacheArrayTable[i].netwokLatencyOfBC:=
              CacheArrayTable[i]. Workload_BC_QueryRes_Time-
              CacheArrayTable[i]. Workload_Query_Time;
          }
       }
}

Procedure Query_WorkloadAndDBandwidth (Query_Workload_Interval) {
  //send query messages to all CSs periodically
  for i:= 1 to NumOfCSs
  {
    CacheArrayTable[i].WorkloadAndDBandwidth_Query_TS+=1;
    CacheArrayTable[i].WorkloadAndDBandwidth_QueryTime=getCurrentTime();
    SendMsg (ICP_QUERY_WORKLOADANDDBANDWIDTH,
               mySwitch.IPAddress,
               CacheArrayTable[i].WorkloadAndDBandwidth_Query_TS,
               CacheArrayTable[i].IPAddress)
  }
  //if query message lost or no response, workload set to infinity, dBandwidth set to 0.
  wait until (getCurrntTime()>sendTime+Timeout)
  for i:=1 to NumOfCSs {
    if (CacheArrayTable[i]. Workload_DBandwidth_QueryResponseTime<
        CacheArrayTable[i].WorkloadAndDBandwidth_QueryTime)
    {
     CacheArrayTable[i].WorkloadOfSC=INFINITY;
     CacheArrayTable[i].DBandwidth=0;
    }
    if (CacheArrayTable[i]. Workload_RC_QueryResponseTime<
        CacheArrayTable[i].WorkloadAndDBandwidth_QueryTime)
     CacheArrayTable[i].WorkloadOfRC=INFINITY;
    if (CacheArrayTable[i]. Workload_RC_QueryResponseTime<
        CacheArrayTable[i].WorkloadAndDBandwidth_QueryTime)
     CacheArrayTable[i].WorkloadOfBC=INFINITY;
  }
}

Procedure RequestClassification (se: SimulationEvent) {
  if (logEntry.contentType==audio || logEntry.contentType==video) se. objType==SC;
  if (logEntry.contentType==E-Commerce) se.objType==RC;
  if  (logEntry.contentType==email || logEntry.contentType==textual ||
        logEntry.contentType==image) se.objType==BC;
}

Procedure DBURouting (se: SimulationEvent) {
  Boolean hit=false;
  for (i:= 1 to NumOfCSs)
   //cache hit--the prefix of the requested object cached in CS_i
    if (CacheArrayTable[i].Content ⊃ req.Content)  {
      hit:= true;
      //disk bandwidth left of CS_i is greater than a prefix space?
      if (CacheArrayTable[i].DBandwidthUnused>= SpaceOfaPrefix)
        //place CSi to CandidateCSArray which contains CS with enough space
        CS_i->CandidateCSArray;
    }

 if (hit==true)
   if (sizeOf (CandidateCSArray)>0) //some CSs have enough DBandwidth
    {
     for (i:= 1 to sizeOf (CandidateCSArray)) {
       ResponseTimeArray[i].ResTime := (CacheArrayTable[i].networklatencyOfSC
        + processTimeOfSC$_{cs}$ (CacheArrayTable[i].WorkloadOfSC);
        if (ResponseTimeArray[i].ResTime < MinResTime) {
          MinResTime := ResponseTimeArray[i].ResTime;
          destinationAddress: = ResponseTimeArray[i]. IPAddress;
        }
      }
        sendTCP_SYN (destinationAddress);
    }
    else //no CSs have enough DBandwidth
    {
      se.requestedObject:=null;
      sendHTTP_RES (clientAddress);
      sendTCP_FIN (clientAddress);
    }
  else //cache-miss
  {
    for (i:= 1 to NumOfCSs)
        if (CacheArrayTable[i].DBandwidthUnused> MaxDBandwidthUnused) {
          MaxDBandwidthUnused := CacheArrayTable[i].DBandwidthUnused;
          DestinationAddress:= CacheArrayTable[i].IPAddress;
        }

     se.requestedObject:=null;
     sendHTTP_RES (clientAddress);
     sendTCP_FIN (clientAddress);

     if (MaxDBandwidthUnused>=Space4aPrefix) sendTCP_SYN (DestinationAddress);
  }//end cache-miss
}

Procedure MRTRouting (se: SimulationEvent) {
  for i:= 1 to NumOfCSs
  {
     if (CacheArrayTable[i].Content $\supset$ req.Content) $F_p$ = cal $F_p$ ();
     else    $F_p = 1$;
     ResponseTimeArray [i]. ResTime=$F_p$ * (CacheArrayTable [i].networkLatencyOfRC
     + processTime * (CacheArrayTable [i]. WorkloadOfRC)
     + 2*RTTA$_{cs\_ws}$ + processTime (WebServer. Workload))
     + (1- $F_p$ )*( CacheArrayTable [i].networkLatencyOfRC
     +processTime*(CacheArrayTable [i]. Workload));

  //pick the cache server with the minimum response time
  for (i:= 1 to NumOfCSs)
     if (ResponseTimeArray[i]. ResTime<MinResTime) {
       MinResTime=ResponseTimeArray[i].ResTime;
       destinationAddress= ResponseTimeArray[i].IPAddress;
     }
  sendTCP_SYN (destinationAddress);
}

Procedure WorkloadRouting (se: SimulaitonEvent) {
   //pick the cache server with the minimum workload
  for (i:= 1 to NumOfCSs)
     if (CacheArrayTable [i]. Workload <MinWorkload) {
       MinWorkload = CacheArrayTable [i]. Workload;
       destinationAddress= CacheArrayTable [i].IPAddress;
     }
  sendTCP_SYN (destinationAddress);
}
```

/*Pseudo code for CS*/

```
/* a CS receives TCP_SYN, HTTP_REQ from switch, sends TCP_ACK, HTTP_RES, TCP_FIN to switch; a CS
sends TCP_SYN, HTTP_REQ to WS, and receives TCP_ACK, TCP_FIN, HTTP_RES from WS a CS sends
content, workload, and dbandwidth update message to switch, receives query_workload, query_dbandwidth,
update_content_ack from switch*/

/* On receipt of TCP messages*/
Procedure onReceiveTCPMessages (msg:TCP, se:SimulationEvent) {
 if (msg.type==TCP_SYN)          //from switch
   sendTCP_ACK (switchAddress);
 if (msg.type==TCP_ACK)          //from WS
    sendHTTP_REQ (WS, ObjType);
 if (msg.type==TCP_FIN)          //from WS
   if (se.objType==SC) { }
    else // (se.objType==RC || se.objType==BC) sendTCP_FIN(switchAddress);
}

/*onReceipt of HTTP messages*/
Procedure onReceiveHTTPMessage (msg:HTTP, se:SimulationEvent) {
 //HTTP_REQ from switch, for CS, RC or BC.
 if (msg.type==HTTP_REQ) {
   if (se.objType==SC) {
     if (DBandwidthUnused>=SizeOfPrefix) {
       DBandwidthUnused:= DBandwidthUnused-SizeOfPrefix;
       if (requestedObj in CS) {
         gotPrefix:=true;
         sendHTTP_RES(switchAddress, prefix);
       }
       else {
         gotPrefix:=false;
         sendHTTP_RES(switchAddress, null);
       }
       DBandwidthUnused:= DBandwidthUnused+SizeOfPrefix;
     }
     else //not enough DBandwidth {
       gotPrefix:=false;
       sendHTTP_RES (switchAddress, null);
     }
     sendTCP_FIN();
   }// end if (se.objType==SC)
 }// end if (msg.type==HTTP_REQ)

 if (msg.type == HTTP_RES) // from WS {
   if (se.objType==SC) { add prefix to CS; }
   else // (se.objType==RC || se.objType==BC ) {
      add object to CS;
      sendHTTP_RES (switchAddress, obj);
      sendTCP_FIN (switchAddress);
   }
 }//end  if (msg.type == HTTP_RES) // from WS
} // end procedure

// send content info to switches periodically
Procedure distributeContent (content: BLOOM_FILTER, num: int) {
 for i := 1 to NumOfSwitches {
   SwitchArray [i]. Content_Update_TS +:=1;
   SendMsg (ICP_UPDATE_CONTENT, myCache.IPAddress, content, num,
              SwitchArray[i].Content_Update_TS, SwitchArray[i].IPAddress);
 }
 sendTime:= getCurrentTime();
 wait until (getCurrentTime()>sendTime+TimeOut)
 for i:=1 to NumOfSwitches do {
  if (SwitchArray[i]. Content_Update_TS
      !=SwitchArray[i].Content_Update_RCK_TS)
    SendMsg (ICP_UPDATE_CONTENT, myCache.IPAddress, content, num,
              SwitchArray[i].Content_Update_TS, SwitchArray[i].IPAddress);
  }
}

Procedure onReceiveMessage (msg:ICPMessage) {
 switch (msg.OPCode)
 case ICP_UPDATE_CONTENT_RCK:
   for i :=1 to NumOfSwitches
    if (SwitchArray[i].IPAddress==msg.SenderAddress)
     SwitchArray[i].Content_Update_RCK_TS:=msg.TS;
  break;

 case ICP_QUERY_WORKLOADANDDBANDWIDTH:
   for (i:= 1 to NumOfSwitch )
    if (SwitchArray[i].IPAddress == msg. SenderAddress) {
     sendMsg (ICP_UPDATE_WORKLOAD_DBANDWIDTH, myCache.IPAddress,
               myWorkloadOfSC, msg.TS, SwitchArray[i].IPAddress);
     sendMsg (ICP_UPDATE_WORKLOAD_RC, myCache.IPAddress,
               myWorkloadOfRC, msg.TS, SwitchArray[i].IPAddress);
     sendMsg (ICP_UPDATE_WORKLOAD_BC, myCache.IPAddress,
               myWorkloadOfBC, msg.TS, SwitchArray[i].IPAddress);
   }
}

/* WS receives TCP_SYN, HTTP_REQ from switch or CS, sends TCP_ACK,
 * HTTP_RES, TCP_FIN to switch or CS
 */
/* On receipt of TCP messages*/
Procedure onReceiveTCPMessages (msg:TCP, se:SimulationEvent) {
 if (msg.type==TCP_SYN)  //from switch or CS
  sendTCP_ACK (senderAddress);
}

Procedure onReceiveHTTPMessages (msg:HTTP, se:SimulationEvent) {
 if (msg.type==HTTP_REQ) {
   if (se.from==switch)        //SC request for left part
    sendHTTP_RES (switchAddress, leftPartOfSC);
   if (se.from==CS)        //SC request for prefix or RC, BC request for object {
    if (se.classType==SC)  sendHTTP_RES (CSAddress, prefix);
    if (se.classType==RC || se.classType==BC )  sendHTTP_RES (CSAddress, object);
   }
   sendTCP_FIN (senderAddress);
 }// end  if (msg.type==HTTP_REQ)
}

 Procedure OnUpdateWorkloadTimeOut () {
  myWorkload:= getWorkload ();
  for (i:=1 to NumOfSwitches)
    sendMsg (ICP_UPDATE_WORKLOAD, myIPAddress, myWorkload,
              SwitchArray [i].IPAddress)
}
```