Design and Evaluation of Request Distribution Schemes for Web-Server Clusters

by

Ramandeep Bhinder

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering
Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

DESIGN AND EVALUATION OF REQUEST DISTRIBUTION SCHEMES FOR WEB-SERVER CLUSTERS

BY

RAMANDEEP BHINDER

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

As the World Wide Web experiences increasing use, server systems are being loaded immensely. In response to the increasing load on the servers, several measures have been taken to improve the performance of Web servers. A cost-effective and scalable solution is to use a distributed system consisting of multiple machines acting as back-end nodes for the server. This approach is both cheaper and more fault tolerant than a single large server.

With the growth of the user community on the Internet, Web servers are also expected to manage their resources so as to provide differentiated and predictable Quality of Service to their clients. Most of the work done on server clusters have focused on partitioning, replicating and aggregating the service components of a cluster so as to provide highly scalable, available and easy-to-manage services.

In addition to providing QoS based service differentiation on clusters, it is also required that the cluster resource (memory, bandwidth, CPU) utilization be maximized. In this thesis, I investigate the problem of providing predictable as well as differentiated QoS in cluster systems, while at the same time trying to maximize the resource utilization of the cluster while meeting service level agreements. In order to determine the major factors which effect the performance of a cluster system, I first compare some of the major request distribution algorithms in terms of achieved throughput, hit ratio, and load balancing. I then use the results obtained to design a QoS aware request distribution algorithm which tries to maximize the overall cluster resource utilization while providing differentiated QoS to different classes of clients. Simulations were carried out to compare the major request distribution algorithms and for the developed QoS aware request distribution algorithm.

Dedication

To my daughter Riya

Acknowledgements

I would like to express my gratitude to all those who gave me the opportunity to complete this thesis. First I would like to thank my supervisor Dr. Muthucumaru Maheswaran whose help, stimulating suggestions, and encouragement helped me all the way through my research. I think he is an ideal supervisor anyone could ever wish for. I would also like to thank Dr. Jeff Diamond of TR*Labs* for ideas and suggestions throughout the research period without which this thesis would not have seen light of day.

I would like to thank TR*Labs* for supporting and funding my research. TR*Labs* provides a wonderful research environment and state-of-the art equipment for doing research. I would like to thank Dr. Jose Rueda of TR*Labs* for providing moral and technical support. I would also like to thank all the other members of TR*Labs* for their help and support in my research as well as in a number of other matters.

I would also like to appreciate the help of my colleagues, especially Vasee for his motivating ideas and helping me at various points in my thesis. I would also like to thank Arindam for reading parts of my thesis and providing me with his valuable comments.

Finally, a special thanks to my elder brother for supporting me morally and financially throughout my studies. Without his support I wouldn't be what I am today. I also thank my wife Anita for her constant support and perseverance. Lastly, but not the least I would like to thank my mother.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet is in a mode of continuous change. It has experienced an explosive growth in the last few years in the number of users and in the number of web-services running on it. The information available on the World Wide Web is also experiencing a massive change, with text-only information sources being augmented with video, voice and very large still images. Also, with the last mile problem being solved with the introduction of high bandwidth connections such as cable modems and DSL, the load on the web servers is increasing day by day, making them more stressed than ever. This causes a number of performance problems like network congestion, slow response time and even denial of service.

A number of approaches have been developed to address these problems. Web browsers have addressed this problem by adding client caches. Caching and replication of content is another approach, whereby the web objects are replicated and cached closer to the requesting clients. This approach can be implemented in two ways: (1) servers push the popular web content out to other replicas or caches, and (2) client requests are cached while passing through client or proxy caches. However, the second approach has a very limited impact [BeA97] as it focuses exclusively on caching at a particular client or

set of clients. A client cache does not reduce traffic to a neighboring client, and similarly a proxy cache does not reduce traffic to a neighboring proxy. The server based push approach is better as it allows the clients to share cache space cooperatively, and, since the servers have a better understanding of the data-access patterns, they can take advantage of the network topology and the file's access history to maximize bandwidth savings and reduce load. But servers must also consider issues like the extent of replication of each file which generally is proportional to that file's popularity, and where each file should be replicated so that clients can reach them efficiently.

Another approach is to make the web servers more powerful. This can be done by upgrading a web server to a faster and larger machine in terms of CPU speed, memory, disk size, and network interfaces. This approach can relieve the server from pressure for a short time, but viewing the current web traffic trend, this does not seem to be a cost-effective or a long-term solution. Also, this approach of improving the performance of a single server is not scalable and has a single point of failure. It will cause interruption in service while the server is upgraded. A solution is required which can keep up with the ever increasing request load on the server, provide scalable services and be a cost-effective solution. One such approach is the use of a distributed system consisting of multiple machines acting as nodes for the server. This approach of providing distributed processing power is both cheaper than a single large server of equivalent processing power, and more fault tolerant. In this system, even if a node or two crashes, the service is not interrupted, as only a small percentage of the serving capacity is lost. But, this approach needs a mechanism to handle the request distribution to the different nodes in the server so as to balance the load among the nodes, as well as to provide a *single system image* (SSI) to the clients i.e., it should appear as a single host to the outside world, so that users can interact with the distributed server as if they were interacting with a high performance server without being concerned about the names or locations of

the individual server nodes.

In order to best utilize the resources of the cluster and to achieve a short response time, good load balancing is necessary in the cluster. Several techniques have been employed to distribute the requests to the back end nodes in the server cluster so as to achieve good load balance. One such approach is to use the Domain Name System (DNS) to support the concept of clusters [BrT95]. In this approach, the DNS server is used to resolve a single host name into multiple IP addresses. The DNS server can either return IP addresses for the host name in a round-robin fashion or can select an IP address randomly from among the set of IP addresses. In this way, successive requests are sent to different nodes in the cluster. Another method uses the Network Address Translation (NAT) standard [EgF94], in which a single IP address spoofs for multiple addresses. Although simple, these approaches have several drawbacks. First, caching of IP addresses in the client browsers and in the secondary DNS servers will result in an unbalanced loading of the nodes in the cluster. Second, if a server node crashes, the DNS server or the NAT will still be translating the address for that node, as there is no automatic way for them to recognize the failure and to stop serving the IP address for that node. These mechanisms thus cause uneven machine utilization in the server cluster. Another way is to use a specialized machine acting as a front end to the server cluster. This front end acts as a single point of contact for the clients, hiding the distributed nature of the server, thus forming a single system image for the server. The front end may employ different policies to distribute the requests to the back end nodes. Some of the existing policies are (1) Round Robin distribution, which distributes the requests to the next back end node in its list in a round robin fashion. (2) Weighted Round Robin distribution, which is the most commonly used technique, and is a pure load balancing technique, where the requests are distributed to the back end nodes in a round robin fashion weighted by some measure of load on different back end nodes. For example, the

CPU and the disk utilization, or the number of active connections in each back end may be used as an estimate of the load.

Request distribution techniques based on URLs are also becoming popular, known as "Content Based Request Distribution techniques". In these techniques, the front end takes into account the service/content requested to make the forwarding decision. These techniques show increased cache hit rates [PaA98] in the back-end's main memory caches, increased secondary storage scalability due to the ability to partition the server's database over different back-end nodes and also allow the ability to employ back-end nodes specialized for certain type of requests (e.g. audio and video).

With the growing size of the user community in the Internet, Web-servers are expected to manage their resources to provide differentiated and predictable Quality of Service (QoS) to their clients. The operating system deployed in servers is inadequate to provide QoS to clients. Also, the clusters don't have the provision to provide differentiated QoS, based on the client community.

In this thesis, I compare some of the major request distribution algorithms, in order to examine the factors which effect the performance of a web cluster system. These performance metrics are the throughput (bytes transferred per second between the cluster and the client), the hit ratio (shows how efficiently the memory cache is being utilized) and the load imbalance (shows how well the load is balanced between the nodes in the cluster) Then, I look into providing both predictable as well as differentiated QoS in cluster systems.

1.1 Why QoS in Cluster systems ?

The current systems running on web servers try to maximize the utilization of the resources in the system while simultaneously providing some fairness between the con-

tending processes. Such systems ensure the fair distribution of CPU resources between the processes. With the increase in the usage of high bandwidth connections like cable modems and DSL, it is often desired to differentiate between clients with respect to the resources allocated within a web cluster to them. This may also be required in the case when organizations wish to provide differentiated and higher guarantees to clients who are paying more as opposed to clients who browse for free. In such situations, it is desirable that a given proportion of server resources be reserved for a class of clients, such that it is guaranteed a minimal level of performance that remains unaffected by the behavior of other classes. Further, in order to maximize the profits or the generated revenue, it is desirable to maximize the utilization of the cluster resources while meeting the service level agreements. Hence, it is also required that in cases where the guaranteed share is not being used by a certain class, it should be free to be used by the other classes.

1.2 Thesis Statement and Contributions

The hypothesis of this dissertation is that current cluster based server systems lack effective facilities for providing QoS to their clients based on their desired bandwidth requirements. The cluster systems should also be able to provide service isolation between the different QoS classes. The goal of this dissertation is to develop a request distribution algorithm to provide both predictable and differentiated QoS to clients. At the same time, it should be able to maximize the utilization of cluster resources including the main memory cache, the throughput (bytes transferred per second) and the total generated revenue.

1.3 Dissertation overview

The rest of the thesis is organized as follows: Chapter 2 presents related work. It discusses several techniques used in web-server clusters for distribution of requests to the back-end nodes. Chapter 3 describes different models and mechanisms currently being used in cluster systems. Chapter 4 describes the request distribution algorithms developed in this dissertation. Chapter 5 describes the simulation setup used to emulate the different request distribution algorithms as well as the proposed quality of service aware request distribution algorithm. Further, it provides the results obtained based on a trace workload. Finally, Chapter 6 summarizes the main results and identifies future directions of research.

Chapter 2

Literature Review

Networks of servers based on clusters of workstations are widely used [FoG97] for various applications such as web-hosting, video on demand services, and storing medical images. These server clusters can be organized in different ways. One popular way of organizing them is to use a specialized node as a front-end to the server cluster. The front-end node is responsible for receiving all the client requests and making decisions about which back-end node should service the request. The back-end nodes are responsible for serving the client requests. Another way of organizing server clusters does not use a dedicated front-end node for distributing the requests to the back-end nodes. In this chapter, I look into some of the related work from academia and industry on server clusters. This study can be categorized into two parts. The first describes *best effort service* based (BES) server cluster systems and the second describes *quality of service* (QoS) based systems.

2.1 BES Based Server Cluster Models

These server cluster models do not aim at providing any QoS to the clients. The main aim in these models is to effectively distribute the requests to the back-end nodes. Some

schemes aim to distribute requests such that the overall performance of the server is improved. The BES cluster systems can be further classified into three types. (1) server cluster systems which use a centralized dispatcher to distribute the requests to the back end nodes, (2) server cluster systems which do not make use of a centralized dispatcher for distributing the requests, or not implement the complete request distributing functionality at the front-end node, and (3) commercially used products.

2.1.1 Centralized Dispatcher Systems

This section examines work which uses a specialized centralized dispatcher generally called a front-end node or a web-switch to steer client requests to the back end nodes in the web-server cluster. The cluster hides its distributed nature by providing a single virtual IP address that corresponds to the address of the front-end node. When a client browser tries to resolve the address for such a site, the Domain Name Server for the web site translates the site address into the IP address of the front-end node. In this way, the front-end node acts as a centralized global scheduler that receives all requests and routes them among the servers of the cluster.

A scheduling policy called *client-aware policy* (CAP) was proposed in [CaC01] for clusters providing multiple services such as static, dynamic and secure information. This approach is different than most other content aware approaches in that it is not limited to sites hosting static content. This approach also does not require any modification to the servers, as dynamic decisions are made at the front-end, by considering only the client request and ignoring the server states. The scheme works by distinguishing the client requests based on their impact on the Web server resources (CPU, disk and network) and assigning each request to a load class. It then assigns the multiple load classes among all the servers so that no single component of a server is overloaded.

A simple content aware request distribution policy known as *locality-aware request distribution* (LARD) is discussed in [PaA98]. In this the front end makes routing decisions based not only on the load on the back-end servers but also depending on the content requested, thus achieving high cache hit rates at the back-end nodes. This approach works by dynamically partitioning the working set of documents over the back-end nodes. When a document is first requested, it is assigned a back-end node by choosing a lightly loaded back-end node. Subsequent requests for that document are assigned the same back-end node unless it is overloaded. In case of overload, a new back-end node is chosen from the current set of lightly noded nodes and the target is assigned to it, thus increasing the number of servers caching that document. The servers caching the same document are called the "server set" for that document. If the same document is requested again, it is assigned to the least loaded node from the server set for that document, unless all the nodes in the server set are overloaded, in which case the server set is further increased as described earlier.

An approach similar to LARD known as the *Harvard Array of Clustered Computers* (HACC) is proposed in [ZhB99]. It is a content-aware technique, which tries to improve performance by making use of locality information and load information at the back-end nodes when making forwarding decisions at the front-end. Their work is different than LARD, in that they focus on static as well as dynamic files. Through prototype implementation they show that their approach has substantial performance enhancement over conventional cluster approaches which do not take into account the content requested while making forwarding decisions.

2.1.2 Distributed Dispatcher Systems

For a cluster based web server, it is desired that the cluster provide a single-system image, so that the distributed nature of the cluster is transparent to the clients. When using a dedicated front-end node, this is accomplished by publishing the IP address of the front-end node which receives all the client requests. When a dedicated dispatcher is not used, or when a distributed dispatcher is used, the issue of how to form a single system-image for the cluster arises. A simple solution as suggested in [BrT95] is to use the *Domain Name Server* (DNS) for the site. When the DNS server receives a translation request, it selects the IP address of one of the back-end nodes in the cluster in a round-robin fashion.

But several problems arise with this method. First, the name-to-IP address mapping may be cached in some intermediate DNS server as there may be several name servers between the client and the DNS server for the site. This caching would result in the sending of bursts of requests from new clients to the same back-end node in the cluster, leading to significant load imbalance. The DNS server can force a mapping to a different server IP every time a request for name resolution for the address comes by specifying a low time-to-live (TTL) value for a resolved name. But, this would result in an increase in traffic on the network for name resolution. Another problem is due to the caching of the resolved mapping at the client, again resulting in load imbalance at the server cluster. Also, in the event of the failure of a back-end node, the DNS server may continue translating the address for that node. A good comparison between the RR-DNS approach and the front-end based approach can be found in [DaK96].

Another technique called ONE-IP was proposed in [DaC97] to publicize a single server name for the entire cluster. In this, the *ifconfig alias* option available on most UNIX platforms was used to provide a common secondary IP address to all machines in the

cluster. All the machines in the cluster have different MAC addresses and different primary IP addresses. When a packet from a client arrives for the assigned secondary IP address, it is broadcasted on the server network as an Ethernet broadcast packet. Filtering is used on each back-end nodes, so that each packet is processed by only one back-end node. The back-end nodes use the secondary IP address to respond directly to the clients.

The ONE-IP technique requires that the router changes the MAC address of all incoming packets to a broadcast address, and also requires a permanent ARP entry at the router, to associate the secondary IP address of the cluster with the Ethernet broadcast address. [VaC01] propose a method called the *Clone Cluster Method* to overcome the need for router configuration in ONE-IP. Unlike the ONE-IP technique, the clone cluster method requires all the back-end nodes to have the same IP and MAC address. All the machines in the cluster are connected to a shared-medium Ethernet LAN, so that all the packets are seen by all the machines. Each machine is given a unique id, which is used by a filter mechanism to accept packets destined for that machine.

An important issue with using a single centralized front-end node to the cluster is limited scalability [DaK96], and single point of failure. The performance of the web cluster cluster should increase with the increase in the size of the web cluster. With a centralized dispatcher, the cluster cannot be scaled beyond the point where the front end node reaches its maximum capacity and becomes a bottleneck. Also, the failure of the front-end could result in non-availability of the cluster service. [BeC98] propose a scalable and fault-tolerant approach called *Distributed Packet Rewriting*, where the task of distributing the client requests is performed by all the nodes in the cluster. The client requests are sent to individual nodes in the cluster, which then forward the request to the appropriate node which would be serving the request. The requests can be sent to the individual nodes in the cluster using some simple approach such as round-robin DNS. To

balance the load on the cluster, routing of requests by the nodes in the cluster could be done in a stateless manner or on per-connection state basis. The routing in the stateless manner uses some type of hashing function to forward the requests to the appropriate node. For per-connection state routing, all the nodes keep track of the connections on the other nodes, and using the state information forward to request to the appropriate node. However, the study does not show any implementation or simulation results for the scalability achieved by the given approach.

[AvB99] uses the approach suggested in [BeC98] to load balance a cluster of web-servers. Through prototype implementation, they show that the per-connection state approach of Distributed Packet Rewriting will achieve better throughput and mean response time than the stateless approach. The requests are distributed to all the nodes in the cluster using Round-Robin DNS. Each machine in the cluster keeps track of the IP addresses and the current load information on all the other machines in the cluster. The load information is broadcast by each machine to the other machines in the cluster at regular intervals of time. When a new request arrives from a client, the nodes use this load information to decide whether the request should to be routed to another node or whether it should be served locally. The nodes use a certain threshold value of load to decide whether to serve the request locally or whether to forward it.

With content-aware request distribution, the problem of the front-end becoming a bottleneck is more severe, as the front-end node must carry out complex forwarding decisions as well as implement either a form of TCP handoff or TCP splicing mechanism. This is because in content-aware request distribution the front-end has to make a TCP connection with the client prior to assigning the request to a back-end node. The front-end becoming a bottleneck could limit the performance improvement and the scalability achieved by the server cluster. [ArS00] present a scalable architecture for Web server clusters implementing content-aware request distribution. They show that conventional

PC/Workstation based front-ends can only be scaled to a relatively small number of server nodes when using content-aware request distribution.

To show the scalability obtained by the new architecture, they separated the front-end into two units. The first unit, called the dispatcher, implements the request distribution strategy, and decides which back-end node should handle a given request. The second unit called the distributor, interfaces with the client and implements the mechanism that distributes the client requests to the back-end nodes. This second unit implements either a TCP handoff or TCP splicing mechanism, and hence is responsible for the bulk of overhead occurring at the front-end. In the proposed architecture the distributor component is separated from the front-end and is implemented in each back-end node, as the tasks performed by the distributor component are completely independent. The front-end still implements the dispatcher component, as the dispatcher typically requires centralized control. The dispatcher component can be implemented as a simple layer-4 switch, and use simple strategies like DNS round-robin to distribute the requests to the distributor components residing in the back-end nodes. The distributor in the back-end nodes may forward the incoming requests to another back-end based on the requested content.

[ChK01] propose a new strategy called *Workload-Aware Request Distribution* (WARD). In this, they have used the cluster architecture as proposed in [ArS00], whereby the front-end just makes forwarding decisions and the TCP handoff operation is carried out by the back-end nodes. Since TCP handoff is an expensive operation, they have tried to minimize the forwarding overhead occurring from TCP handoff by identifying a small set of most frequent files, called *core*. The files forming the core can then be processed by any server in the cluster, while the rest of the files are partitioned to be served by different cluster nodes. For identifying the optimal size of the core, they have designed an algorithm called *ward-analysis*. The ward-analysis algorithm uses the workload char-

acteristics such as the frequency of access of files and the sizes of individual files, and system parameters such as number of nodes in a cluster, forwarding overhead, and disk access times, to generate the optimal core. In addition to using the architecture proposed in [ArS00], they have also used a modified architecture, whereby the forwarding decisions and the TCP handoff operations are carried out by the back-end nodes. This was done to reduce the overhead and delay occurring due to querying of the front-end to make the routing decisions. The routing decisions at the back-end nodes are made on the basis of previous days access patterns.

With content-aware distribution, the major cause of overhead is the setting up of a TCP connection with the front-end, which can thus easily become a bottleneck. To avoid the overhead due to the TCP connection at the front-end, while still using content-aware distribution, a new policy was proposed in [ChD01] and [ChL00] called *FLEX*. This approach was suggested for servers which are used to host multiple web sites as a set of virtual servers.

FLEX works by partitioning the web sites into a number of groups depending on their memory and load requirements and then assigning each group to a node in the cluster. It uses the working set sizes and access rates of different sites from the web site logs to get a measure of the memory and load requirements. It then uses the DNS server for that site to do the routing to the nodes in the cluster. After partitioning the web sites on different nodes, the DNS server is set up with the corresponding information by mapping all the sites on a particular node to the IP address of that node. Sites which cannot fit on a single node are replicated onto more than one node and Round-Robin DNS is used to route requests for that site to the nodes hosting that site. To adapt to changes in traffic patterns for different sites over time, the request access patterns are monitored and changes are applied to the groupings on a regular basis. This solution is very simple, as it does not require installation of additional hardware, and is also scalable as it does not

use a centralized front-end node. Through simulation it was shown that it could achieve a performance improvement of about 110% to 250% in server throughput as compared to the conventional Round-Robin scheme. However, the scheme was not compared to other content-aware schemes like LARD. Also, like most other content-aware schemes it can only work with sites hosting static content.

2.1.3 Commercial Products

There has been a lot of research on cluster servers, and many of those ideas have been developed and implemented as commercial products to be used with cluster based servers. [HuG97] describes a product developed by IBM called *Network Dispatcher*. The Network Dispatcher is a TCP connection router which could be used as a front-end node to a server cluster to achieve load sharing across the nodes in the cluster. The back-end nodes form a virtual server by providing services on a single IP of the Network Dispatcher. Packets are forwarded to the back-end nodes using the Weighted Round Robin connection allocation algorithm, whereby the back-end nodes are assigned some weight depending on the load on them. Packets are then forwarded to the back-end nodes depending on their assigned weights. When forwarding the client packets to the back-end nodes, the Network Dispatcher modifies the packets by replacing the IP address of the virtual server with the IP address of the chosen back-end node. When sending the response back to the client, the back-end nodes provide the IP address of the virtual server. Hence, the response packets can be sent directly to the client, bypassing the Network Dispatcher. Since the response packets from the server are generally larger than the packets from the client, the Network Dispatcher provides some performance improvement over methods where the two way traffic flows through the front-end.

Another product developed at Berkeley called *The Magicrouter* [AnP96], uses a

packet-filter based approach. This approach emphasizes on reducing the packet processing time at the front-end node by using an approach called "Fast Packet Interposing". This approach allows a user level process to run nearly as fast as when implemented at the kernel level. The Fast packet interposing approach shares memory between the user space and the kernel space, thus eliminating copying between the kernel and user space. The Magicrouter works as a front-end for a web server cluster. When a packet arrives from the client, the Magicrouter translates the network address depending on the chosen back-end node. Similarly, the address translation is done for the response packet, as it flows through the Magicrouter, unlike the Network Dispatcher.

An architecture similar to the Network Dispatcher, also developed at IBM, is called *TCP Router* [DaK96]. The TCP Router is also used as a front end node to forward requests to the individual back-end nodes in the cluster. The TCP Router is different from Magicrouter in that it forwards the response packets directly to the clients, thus eliminating the need for rewriting response packets at the TCP Router. Unlike the Magicrouter, the TCP Router looks into the issue of load balancing the back-end nodes in the cluster. For load balancing, the TCP Router keeps track of the connection assignments and assigns connections to the back-end nodes depending on their current state.

To allow for response packets to be sent directly to the clients, the Network Dispatcher and the TCP Router require modification in the back-end server kernels, which is not needed with the Magicrouter approach. Another product is the [LoC96] Cisco *LocalDirector*, which also rewrites the TCP/IP headers at the front-end of all packets flowing between clients and servers in both directions. The LocalDirector offers various dispatching policies such as the fastest response algorithm that dispatches the request to the server that was fastest in responding to the previous connection requests, and the least connections algorithm, that forwards the incoming requests to the server with the least number of active connections. This product is also able to support some stateful

services, such as SSL through the use of a sticky flag. This is done by forwarding multiple requests from the same client to the same server within a period of time.

A content-aware product developed at Nortel Networks is called as the *WebOS SLB*, and is based on a TCP splicing approach. This product establishes a TCP connection with the client and the target server, and then splices the connection between them. To do so, it modifies the TCP header of every packet that travels between the client and the server and recalculates the TCP/IP header checksums. To prevent the front-end node from becoming a bottleneck, this product relies on a dedicated network processor integrated with an operating system. The WebOS SLB partitions the working set over the back-end nodes such that each server stores specific documents. When a specific document is requested, the front-end node assigns the request to the target back-end node.

Another content-aware product called the *scalaserver* was developed at the Rice University. This product is based on TCP handoff and allows the response packets to be sent directly to the clients. The front-end node distributes incoming requests according to the LARD policy described earlier [PaA98]. After the connection is handed over to the selected back-end node, the node then communicates directly with the client. Incoming traffic on an already established connection is forwarded to the target back-end node through an efficient forwarding module layered at the bottom of the front-end node's protocol stack. This product requires that the server operating system be modified in order to support the TCP handoff protocol. This product also supports HTTP/1.1 persistent connections and is able to assign requests in the same connection to different back-end nodes with the use of a back-end forwarding mechanism [ArD99]. Other similar products are the *ACEdirector*, developed by Alteon Websystems [AlT02], the *ServerIron* [FoU02] developed by Foundry Networks, and the [ReS96] *Resonate Dispatch*, which are all capable of content-aware switching where the URL is examined before transferring

the connection to the back-end node that can serve the request.

2.2 QoS Based Server Cluster Models

Recent studies on providing QoS support [[JoL95], [PaB98], [BrB99], [BrG98], [BaD99]] have mostly focused on single node web servers. As described in the last section, most of the work on cluster servers have focussed on partioning, replicating and aggregating the service components of the cluster so as to provide highly scalable, available and easy-to-manage services. Limited studies have been conducted on providing service differentiation for cluster-based servers. There is also a lack of comprehensive QoS support for these Cluster-based servers.

[ArD00] introduce a facility called *Cluster Reserves* for ensuring performance isolation between different service classes hosted on the cluster while enabling high utilization of the cluster resources. Cluster reserves builds upon the work of *resource containers* [BaD99] used for performance isolation in a single node web server to achieve cluster-wide performance isolation. It uses a cluster resource manager for partitioning the resources allocated to a cluster reserve to individual nodes in the cluster. To compute the partition, the cluster resource manager collects resource usage statistics from the cluster nodes and maps the allocation problem to an equivalent constrained optimization problem. The method used for performance isolation is independent of the request distribution strategy deployed in the cluster. This approach requires significant changes to the operating system kernel at the front-end and at the back-end nodes.

[ZhT01] propose a flexible mechanism called *Demand-Driven Service Differentitation* for specifying service quality classes and a scheduling algorithm that dynamically adjusts server partitions based on request resource demands. They present a scheduling algorithm which decides for each period, how many servers should be assigned to a request class

and what percentage of requests should be dropped from the class in order to provide service differentiation and performance guarantees. The algorithm tries to provide better services to higher priority classes, especially when the system is overloaded. At the same time care is taken so that requests from the lower priority classes are not oversacrificed for requests from higher priority classes. The limitation with this scheme is that it partitions all replica's for each service into several groups and each group is assigned to handle requests from one service class. Thus, it may not respond promptly to changing demand, as the partitioning cannot be done very frequently.

A framework is proposed in [ShT02] for providing class-based service differentiation in terms of resource allocation and admission control. They introduce a class-aware load balancing scheme called *class LB* to ensure a balanced distribution of service requests to a set of replicated service nodes, and a multi-queue scheduling scheme for producing high QoS yield (economical benefit) at each service node. The goal of the scheduling algorithm is to provide guaranteed system resources for all service classes and schedule the remaining resources to achieve high aggregate QoS yield.

2.3 Miscellaneous

URL-aware distribution requires the migration or redirection of TCP connections from the front-end node to the back-end node which would be serving the client request. One approach for redirection is to use the front-end as a *TCP gateway.* In this approach the front-end node makes two TCP connections: one with the client and one with the back-end node. The front-end then passes the client requests to the back-end node on the TCP connection with the back-end node. The response received from the back-end node is transferred to the client using the TCP connection with the client. For the connection transferring between the two TCP connections an application layer process is used, which

produces a lot of overhead at the front-end.

To improve the performance of the front-end based switches acting as TCP gateways, *TCP splicing* [MaB98] was proposed. This approach uses a Kernel level process for the data forwarding operations between the two TCP connections, thus reducing the overhead caused due to packets traversing up the protocol stack to the application layer and then down again. This approach works by doing the appropriate address translation and sequence number modification on the packets. The packets arriving from the clients are modified so that the addresses and sequence numbers on these packets match the ones that would be found on the corresponding packets if the front-end node received the packets and put them back on the TCP connection from the front-end node to the back-end node. Similarly, the address and sequence number modification is done on the packets received from the back-end nodes. [CoR99] use the TCP splicing approach to design and implement a layer-7 switch that supports URL-aware redirection of HTTP traffic.

The TCP splicing approach performs better than the TCP gateway approach but still incurs high overhead because all the response packets from the server must be forwarded via the front-end node. A method called *connection hand-off* is discussed in [HuN97] for migrating the TCP connection from the front-end to a back-end node in the cluster. This method enables the forwarding of the server response directly to the client without passing through the front-end node. This is done by handing off the TCP connection established with the client at the front-end to the back-end node which would be serving the given client request. It enables a connection hand-off by adding an option to the TCP protocol specification. When a client request first arrives, the front-end makes a TCP connection with the client through the three-way TCP handshake. After the handshake, the front-end node sends a SYN packet to the back-end node. The SYN packet contains the front-end node's address as the source IP address and an option with the IP address

of the client and the initial sequence number selected by the front-end. The back-end node then goes through a three-way handshake with the front-end node using the passed sequence number. Then the back-end node responds directly to the client using the passed client's IP address. A similar approach is proposed in [PaA98].

Another important issue with cluster based servers employing content-aware request distribution is the support of HTTP/1.1 persistent connections. With HTTP/1.1, multiple requests may arrive on a single HTTP connection. Hence, content-aware distribution which distributes the requests at the granularity of TCP connections may forward all the requests arriving on that persistent connection to a single back-end node. This may lead to the overloading of some of the back-end nodes.

[ArD99] describes two mechanisms to achieve efficient content-aware request distribution in the presence of HTTP/1.1 persistent connections. The TCP handoff techniques described above can support HTTP/1.1 connections, but all requests must be served by the back-end node to which the connection was handed off originally. The first mechanism called *multiple handoff* is an extension of the previously described HTTP handoff mechanisms. In this method, the front-end node is modified to support HTTP/1.1 connections by allowing it to migrate connections between the back-end nodes. The second mechanism described is called *back-end request forwarding.* In this approach, the front-end hands off the request to the appropriate back-end node using the handoff mechanisms described above. When the back-end node receives a request which cannot be served by it, the node requests the service from another appropriate back-end node and forwards the response directly to the client. Through prototype implementation it was shown that with the back-end forwarding mechanism, the content aware distribution policy achieves up to 26% better performance with persistent connections than without the mechanism.

Chapter 3

Cluster Models

A server cluster or web cluster refers to a web site that uses a set of server nodes housed together at a single location to serve client requests. Usually the web cluster uses a single hostname to provide a single system image and a single interface to the users. The server cluster uses a specialized front-end node to provide a single virtual IP address and to distribute client requests to the back-end nodes. This front-end node hides the distributed nature of the server from the clients and may be used to employ different request distribution algorithms. A simple implementation of a server cluster is shown in Figure 3.1.

As explained in Chapter 2, the front-end node could be centralized or may be distributed, depending on the policy used for request distribution. In this chapter I will focus generally on the front-end's request distribution mechanism, without worrying about it being centralized or distributed. We will call the front-end node a front-end switch in general from here on. This front-end switch could be implemented on some special purpose hardware or could be software running on some operating system.

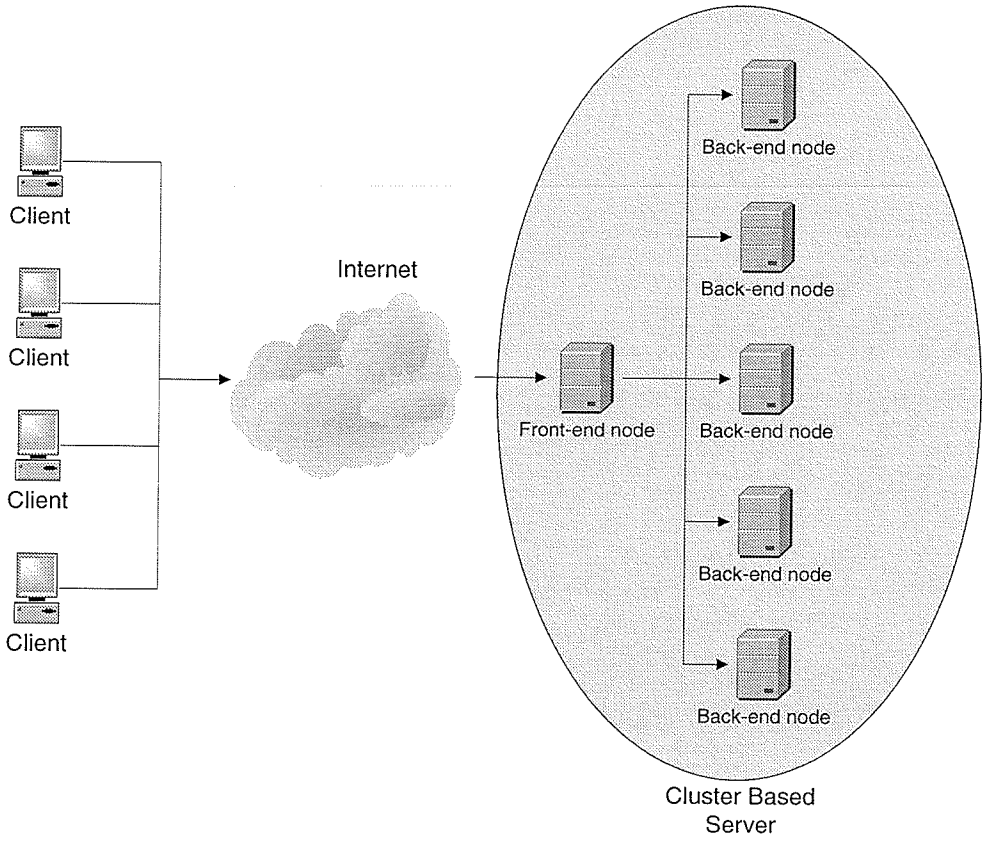

Figure 3.1: Cluster Based Server Architecture

3.1 Request Distribution Mechanisms

To uniquely identify the back-end nodes in the server cluster, the front-end switch could either use the IP address or MAC addresses of the back-end nodes, depending on the policy used. Hence, depending on whether the front-end switch uses IP address or MAC address of the back-end nodes to steer client requests to the target node, the front-end node can be classified as a *layer-4* or *layer-7* switch. This classification is made according to the OSI protocol stack layer at which the front-end switch works.

3.1.1 Layer-4 Switches

At layer-4 of the OSI protocol stack (TCP/IP level), I do not have access to the content requested in the packet. So the dispatching policies using layer-4 switches perform content-blind routing. Since layer-4 switches work at the TCP/IP level, they select the target server for a client request upon the arrival of the first TCP SYN packet from the client. These switches uses a binding table to manage the TCP connection assignments for the client requests. The binding table could contain the information available at the TCP/IP level i.e., the IP source address, the source port, the IP destination address, the destination port, and some other relevant information to be used by the dispatching algorithm.

When the front-end switch receives a packet from the client, it examines the bits in the flag field of the header to determine whether the packet is for a new connection, an already established connection, or none of them. If an incoming packet is for a new connection, i.e. has the SYN flag bit set, the front-end switch selects a target back-end node using the dispatching policy being used. It also inserts a record of the connection-to-back-end node mapping in the binding table and routes the packet to the selected back-end node. If the incoming packet does not have its SYN flag bit set, i.e., is not for a new connection, the front-end switch looks up the binding table to check whether the packet belongs to an existing connection or not. If it belongs to an existing connection, the front-end switch routes the packet to the back-end node handling that connection. Otherwise, the front-end switch just drops the packet.

The layer-4 front-end switches can further be classified depending on the mechanism used to route packets coming from the clients (incoming packets) for the back-end nodes and packets coming from the back-end nodes (outgoing packets) for the clients. Mainly the classification is done on the basis of the outgoing packets, i.e. the server to client

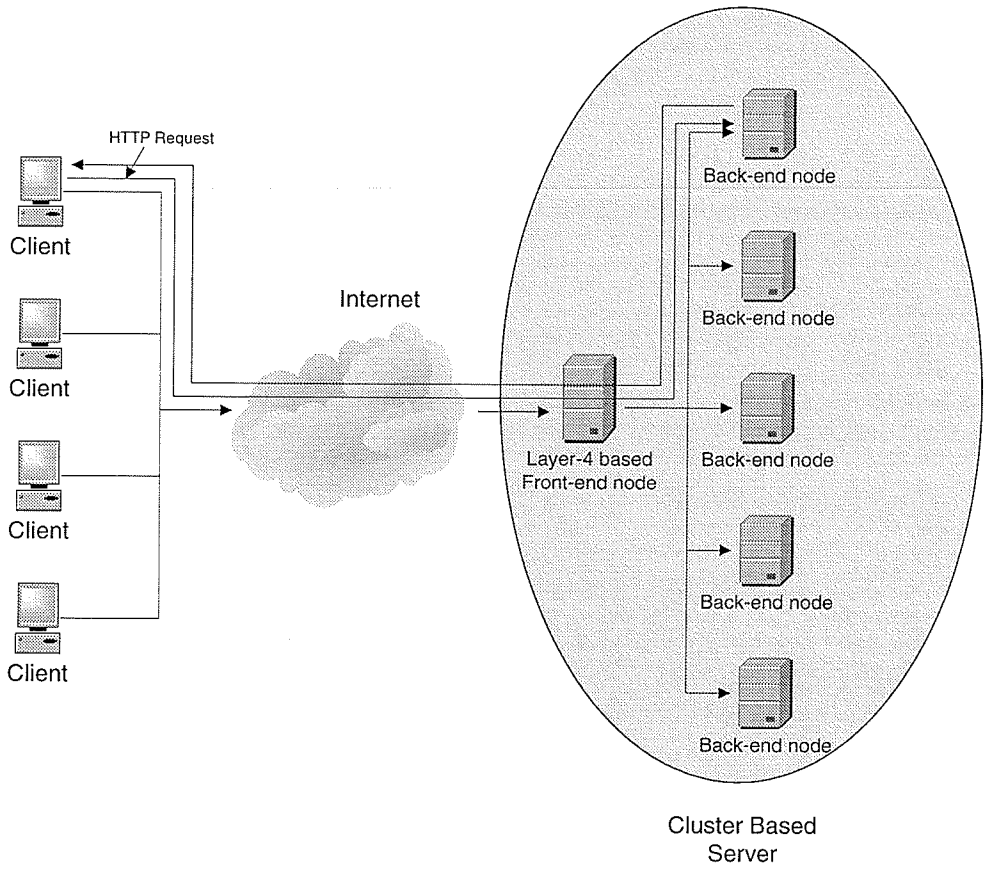

Figure 3.2: Layer-4 bidirectional architecture

packets. If the incoming and outgoing packets pass through the front-end switch, I call the architecture a *bidirectional/two-way* architecture. And, if only the incoming packets pass through the front-end switch, I call the architecture a *unidirectional/one-way* architecture.

In *bidirectional/two-way* architectures both the incoming and outgoing packets are rewritten by the front-end switch as they pass through the switch as shown in Figure 3.2. In the two-way architecture each back-end node in the cluster is assigned a unique private IP address. The front-end node is assigned a virtual-IP address, which is publicized to the

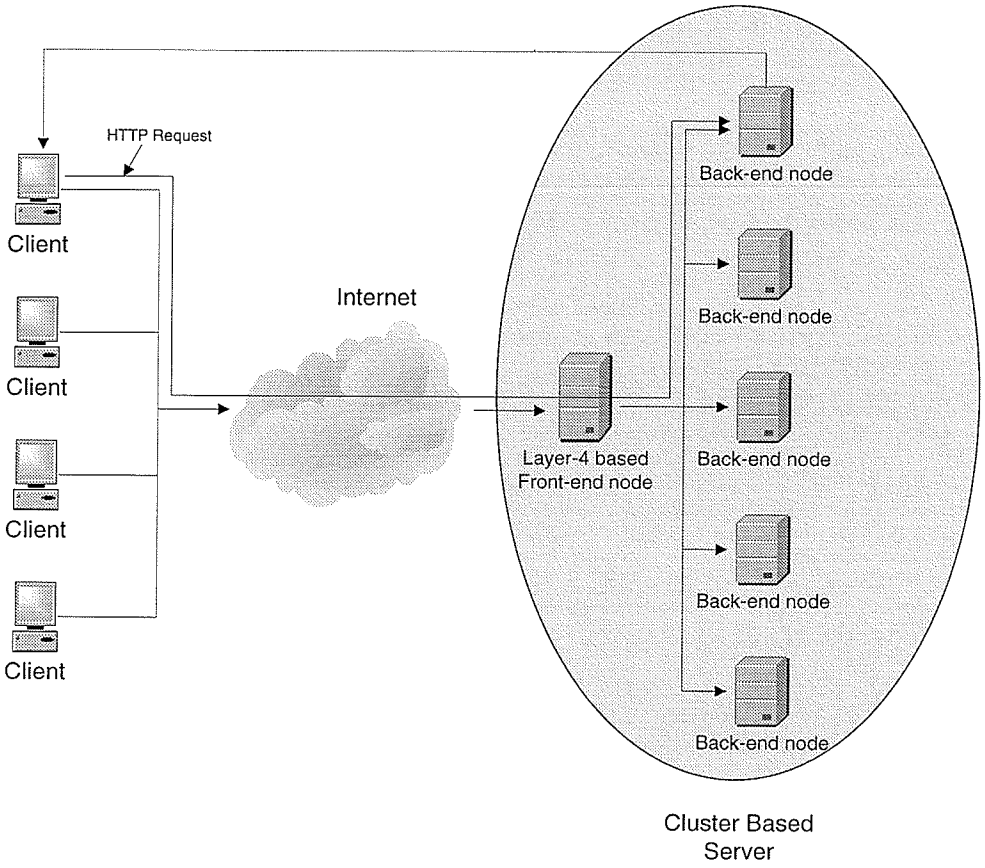

Figure 3.3: Layer-4 unidirectional architecture

clients. When the front-end receives a request from the client, it uses Network Address Translation (NAT) [EgF94] to change the virtual-IP address to the private IP address of the selected back-end node in the destination address field of the packet header. It also recalculates the TCP and IP header checksums in the packets. The response packets from the back-end nodes contain the IP address of the node serving the client request. Hence, the front-end switch needs to change the IP address of the back-end node with its virtual-IP address, and also recalculate the TCP and IP header checksums.

In the *unidirectional/one-way* architecture, only the incoming packets pass through

the front-end switch. The outgoing packets are sent directly to the client by the back-end node serving the request. An example of an unidirectional/one-way architecture is shown in Figure 3.3. When the front-end switch receives a packet from the client, it could use several mechanisms to forward the client requests to the back-end nodes in the cluster. It could use address translation (NAT) called packet rewriting to modify the IP destination in the packet and recalculate the TCP/IP checksums of the incoming packets. It could use IP tunneling [SiM95] to encapsulate each packet within another packet, or it could also forward packet at the MAC level. This is known as packet forwarding.

To route packets using packet rewriting, the front-end switch rewrites the destination IP address of each incoming packet. For this, each back-end node is assigned a private IP address and the front-end switch is assigned a virtual-IP address which is publicized to the world. When the front-end switch receives a packet from a client, it replaces the virtual-IP address in the incoming packet with the IP address of the selected back-end node in the cluster. It then recalculates the IP and TCP header checksum for the packet. To send the response directly to the client, the back-end node replaces its IP address with the virtual-IP address assigned to the front-end switch, and recalculates the IP and TCP and IP header checksums.

IP tunneling is a technique which encapsulates the original IP datagram with its header retained as it is, in another IP datagram by adding an outer IP header before the original IP header. The source and destination address in the outer IP header identify the endpoints of the tunnel, and the source and destination address in the inner IP header identify the original sender and recipient of the datagram. To use IP tunneling for routing, the front-end is assigned a virtual IP address. The user requests the services provided by the server cluster using the virtual IP address. When a user request arrives at the front-end switch, the switch selects a back-end node for servicing the request according to the connection scheduling policy being used. It then adds the entry into

the hash table which records the connection assignments. The front-end switch then encapsulates the packet within an IP datagram and forwards the request to the chosen back-end node. When the front-end switch receives a packet which belongs to an already assigned connection, it checks the hash table for the back-end node serving the request, encapsulates the packet and forwards it to the corresponding back-end node. When the back-end node receives the encapsulated packet, it decapsulates the packet and processes the request. It then returns the response directly to the client according to its own routing table. After the connection terminates or timeouts, the connection record will be removed from the front-end switch's hash table.

The packet forwarding approach assumes that the front-end switch and the back-end nodes in the cluster are on the same local area network. For this approach also, the front-end switch is assigned a virtual IP address, which is used by the clients to access the services provided by the server cluster. The back-end nodes are assigned two IP addresses. A common secondary IP address which is the same as the virtual IP address of the front-end switch is assigned to all the machines in the cluster using the *ifconfig alias* option available on most UNIX platforms. All the machines are also assigned different MAC addresses and different primary IP addresses. When a client request arrives, it is received by the front-end switch as the back-end nodes have disabled the Address Resolution Protocol (ARP) mechanism to prevent collision. The front-end switch then forwards the request to the chosen back-end node using the MAC address of the back-end node. It rewrites the physical address in the packet with the MAC address of the chosen back-end node and retransmits the frame on the network. The front-end switch does not modify the TCP/IP header in the packet. When the back-end node receives the forwarded request, it processes it as it shares the virtual IP address, and returns the response directly to the client.

3.1.2 Layer-7 Switches

Layer-7 switches are more complex and intelligent then the layer-4 switches because they have to inspect the requested content in the HTTP request packet before making any forwarding decisions. Layer-4 switches can select the target back-end node as soon as they receive the initial TCP SYN packet, but with layer-7 switches, the front-end switch must first establish a TCP connection with the client before it can receive the HTTP request from the client. Similar to the layer-4 switches, the layer-7 switches can be classified on the basis of the mechanism to send the packets to the back-end nodes and from the back-end nodes to the clients. If the incoming and outgoing packets pass through the front-end switch, I call the architecture a *bidirectional/two-way* architecture. And, if only the incoming packets pass through the front-end switch, I call the architecture a *unidirectional/one-way* architecture.

In a bidirectional/two-way architecture, the request packets from the clients and the response packets from the back-end nodes pass through the front-end switch as shown in Figure 3.4. To route the client requests to the back-end nodes in the cluster, the front-end switch could use the TCP gateway approach or the TCP splicing [MaB98] approach.

In the TCP gateway approach, the front-end switch maintains TCP persistent connections with all the back-end nodes in the server cluster. The front-end switch uses an application level proxy to forward the client requests to the back-end nodes and to send the response back from the back-end nodes to the clients. When the front-end switch receives a client request, the application level proxy running on it forwards the client request to the target back-end node through the corresponding TCP persistent connection. When the response arrives from the back-end node, the front-end switch forwards it to the client through the other connection with the client.

However, the TCP gateway approach causes a lot of overhead, because of the packets

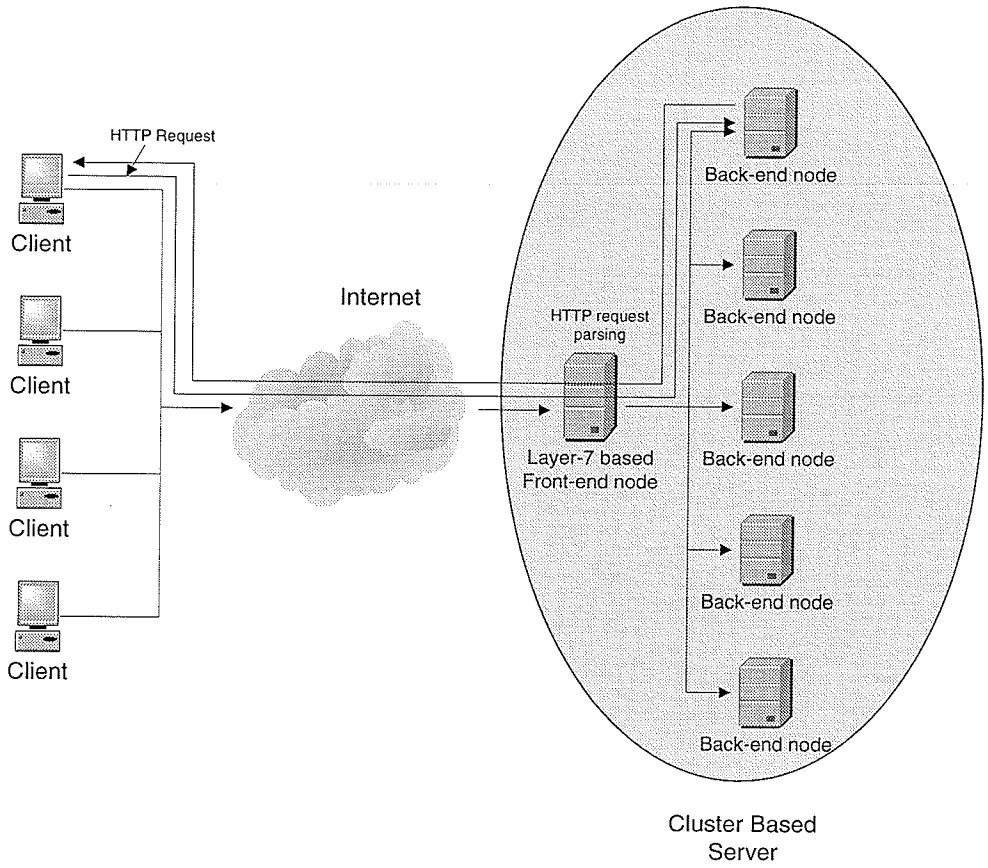

Figure 3.4: Layer-7 bidirectional architecture

traversing up the protocol stack to the application layer and then back again. The TCP splicing approach aims at improving the performance of the TCP gateway approach by using a Kernel level process for data forwarding operations between the two TCP connections. When the front-end switch receives a client request, it completes the TCP connection with the client and chooses the target back-end node and the persistent TCP connection with the back-end node. The front-end switch then splices the two connections using the kernel level process. It does this by modifying the address and sequence numbers in the packets, such that the packets are recognized by the back-end node and the client

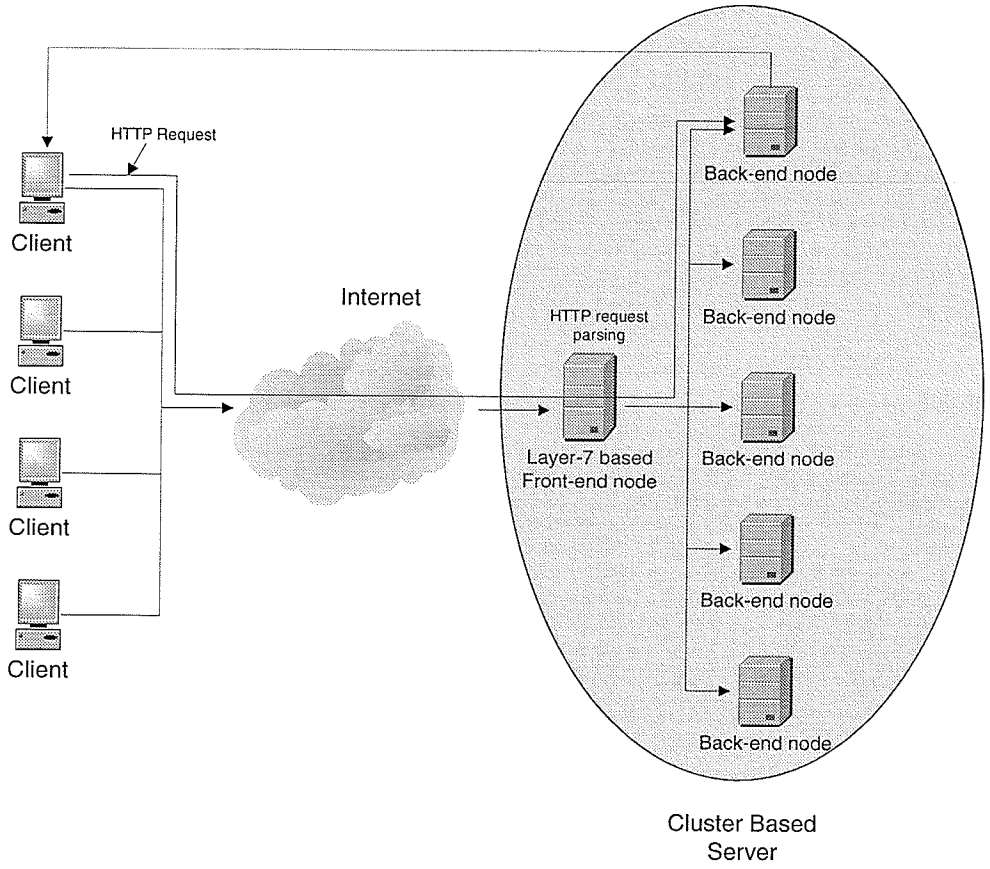

Figure 3.5: Layer-7 unidirectional architecture

as been destined for them. Thus, TCP splicing forwards the packets from the client to the back-end nodes and from the back-end nodes back to the client without having to cross the TCP layer up to the application layer on the front-end switch, thus improving performance.

In the *unidirectional/one-way* architecture the response packets from the back-end nodes are sent directly to the clients, bypassing the front-end switch as shown in Figure 3.5. This relieves the front-end node from the burden of processing the response packets from the back-end nodes. Two approaches can be used by the front-end switch using this

architecture to forward the client requests to the back-end nodes in the cluster.

The first approach is called *TCP connection hop* and was developed by Resonate [ReS01]. In this scheme, the TCP connection is hopped from the front-end switch to the selected back-end node in the cluster. The TCP connection hop is implemented through a software component called Resonate Exchange Protocol (RXP). The RXP is installed on every machine in the cluster. When the front-end switch receives a client request, the RXP encapsulates the entire IP packet as it was sent from the client, in an RXP packet, and sends it to the selected back-end node. When the back-end node receives this packet, the RXP running on it sees that it is an RXP packet and strips off the RXP header. It then sends the original IP packet to the server running on the back-end node. The web server can then send its reply directly to the client, without having to pass through the front-end switch.

The second approach is called the *TCP handoff* approach [PaA98], where the front-end switch hands off the established connection with the client to the target back-end node in the cluster. The handoff protocol runs on the front-end switch as well as the back-end node on top of TCP. It thus requires the modification to the operating system of the front-end switch as well as the back-end nodes.

3.2 Load-Distribution Algorithms

Load-distribution algorithms are used by cluster-based system to distribute the load amongst the server nodes in a cluster. The system should decide the best node to execute any job regardless of where the job originated. It may also decide to transparently migrate some jobs during their execution. These algorithms allow the user to access any resource in the cluster-based server system without worrying about its physical location. The goal is to create the illusion in the minds of the users of a single powerful timesharing system,

rather than a collection of independent but connected machines.

3.2.1 Load Sharing and Load Balancing

The main aim of these algorithms is that the machines in the cluster should share the total processing load requirement of all the users so that the load distribution is more uniform among the machines in the cluster. This results in an improvement in the system performance as seen by the users. These algorithms could either use *load sharing* or *load balancing* schemes for load distribution in the cluster [ShK92]. Load sharing requires the system to select the best machine for executing a newly arrived client request and transparently transfer the job to the selected machine. Load balancing is a strict form of load sharing wherein the system tries to balance the load on all machines by migrating jobs between machines, possibly during the job execution.

The load balancing scheme is more complex than the load sharing scheme, as the former requires the system to check the state of a process already in execution and then transfer this state to the target machine. A load balancing policy tries to ensure that every machine in the cluster does the same amount of work at any point of time. Algorithms for load balancing rely heavily on the assumption that the information available at each back-end node is accurate, in order to prevent jobs being endlessly circulated about the system without making any progress. These schemes are mostly suitable in homogeneous systems as it will be much easier to transfer the state information in such systems. The load sharing scheme tries to send new requests to lightly loaded back-end nodes, and hence distribute the overall load of the system by submitting requests to nodes on the cluster when they are initially requested. The load sharing scheme can be easily used for heterogeneous systems as it does not require communication of any state information between nodes. Thus, the overhead incurred in load balancing algorithms

due to the expensive state transfer may trade off any performance improvement gained by the scheme.

3.2.2 Policy Components

The load sharing scheme can be separated into four components depending on the functionality of the policy, namely: the transfer policy, the selection policy, the location policy and the information policy [ShK92]. The *transfer policy* decides whether a node is suitable to participate in request transfer, either as a sender or as a receiver. Most of the load sharing schemes use a threshold based policy to decide whether to transfer a request to another node or not. In the *threshold based policy*, a node is designated as a sender if its load exceeds a certain threshold value T_1, or as a receiver if the load on the node falls below some threshold value T_2 where $T_2 \leq T_1$. Another transfer policy used is called the *relative transfer policy*. In this policy the load of the node is considered in relation to the loads on the other nodes of the cluster. For example, this policy could consider a node to be a suitable receiver if its load is lower than some other node in the cluster by at least some fixed value δ, or even if its load is the lowest among all the nodes in the cluster.

Once the transfer policy decides that a node is a sender, the *selection policy* selects a request for transfer. If the selection policy fails to find a suitable process, it is no longer considered as a sender. One simple policy used by the selection policy is to select one of the newly initiated requests as the one to be transferred. For request migration, the selection policy should select a request which is small so that it incurs minimal overhead, long lived so that it compensates for the overhead incurred in transferring, and likely to make minimal number of location-dependent system calls.

The *location policy* finds a suitable partner (sender or receiver) for a node, which

has been declared as a sender or receiver by the transfer policy. One commonly used *decentralized policy* to find a suitable node is called *polling*. In polling, a node polls another node to determine if it is suitable for load sharing. Polling could be done serially or in parallel. Polling in parallel is done by using multi-cast or broadcast message passing. Polling could also be done randomly, on the basis of information collected during previous polls, or on a nearest neighbor basis. Another option is to broadcast queries, so that a node broadcasts a message and the nodes available for load sharing reply back, from which one is selected. A *centralized policy* can also be used, where a node contacts a specified node called a coordinator. The coordinator then selects a peer for the requesting node and informs both the nodes.

The *information policy* gathers information about the state of the system to be used by the distribution policy. This policy has to decide what state information about the other nodes in the system has to be collected, where the information has to be collected, and when to collect the information. The information policy can be further classified into three types: demand-driven policy, periodic policy, and state-change-driven policy. In the *demand-driven* policies, a node gathers information about other nodes in the cluster only when needed, for example when it becomes a sender or a receiver. Demand-driven policies can be initiated by the sender (sender-initiated policy), where the sender looks for a suitable receiver to transfer its load to. It could be initiated by the receiver (receiver-initiated policy), where the receiver looks for loads from senders. It could also be initiated by both the sender and the receiver (symmetrically-initiated policy). In the *periodic policy* a node collects information about the state of the other nodes in the cluster periodically. This policy does not adapt their information gathering rate to the system state. In the *state-change-driven policy*, the nodes disseminate information about their states whenever their state changes by a certain degree. The state-change-driven policies could either send the state information to a dedicated centralized point, or they

could send state information to each other.

3.2.3 Dynamic, Static, and Adaptive Algorithms

The load-distribution algorithms can also be classified as dynamic, static and adaptive algorithms [GuB99]. *Dynamic* load-distributing algorithms use system-state information such as load at the back-end nodes to make distribution decisions, while *static* algorithms make no use of such information. Decisions are hardwired in static load-distribution algorithms using some priori knowledge about the system. The *adaptive* load-distributing algorithms are a special case of dynamic algorithms. The adaptive algorithms can dynamically change their load sharing policy as well as their policy parameters on the basis of the changing system state or workload conditions. A simple example of an adaptive algorithm is where I have a distribution policy which performs better than others under certain conditions, while another policy performs better under other conditions. A adaptive algorithm could choose between the two policies based on observation of the current system state. Due to the complexity of the adaptive algorithms they are not widely used. Therefore, the real alternative is among the static and the dynamic algorithms.

Content-blind and Content-aware Algorithms

Since the static distribution algorithms do not consider any state information while making forwarding decisions. The front-end switches that use static distribution algorithms mostly work at the TCP/IP level, as they have enough information at that layer for making forwarding decisions. Hence, the static distribution algorithms can also be called *content-blind static* algorithms, as they do not consider the content requested to make forwarding decisions . On the other hand, the dynamic distribution algorithms, being more complex, can use the system state information to outperform the static distribut-

ing algorithms. The dynamic distribution algorithms could use load information from the back-end nodes to make forwarding decisions. Load information at the back-end nodes could be measured by various metrics such as the processor queue length, the average processor queue length over some period of time, the amount of memory available, the system call rate, the CPU utilization, etc. Since the front-end switch has to make hundreds or thousands of forwarding decisions per second, it should be noted that the metrics used by the dynamic distribution algorithm should be simple and easy to compute [GuB99]. The front-end switch cannot use highly sophisticated algorithms because it could cause the front-end node to become a bottle-neck for the cluster system and perform even worse than some simple static distribution algorithms.

The information used by the dynamic algorithms to make forwarding decisions also depends on the OSI protocol stack layer at which the front-end switch using the algorithm works. If the front-end switch using the algorithm works at the TCP/IP layer, I call it a *content-blind dynamic* algorithm, and if the switch works at the application layer I call the algorithm a *content-aware* algorithm, since it has access to the requested content. From here on I will classify the distribution algorithms as content-blind and content-aware algorithms. In summary, content-blind algorithms can be classified into static and dynamic algorithms, and content-aware algorithms as dynamic algorithms. Figure 3.6 shows the taxonomy for the distribution algorithms.

3.2.4 Content-blind Algorithms

This section describes the main content-blind static and the content-blind dynamic distribution algorithms as shown in Figure 3.7.

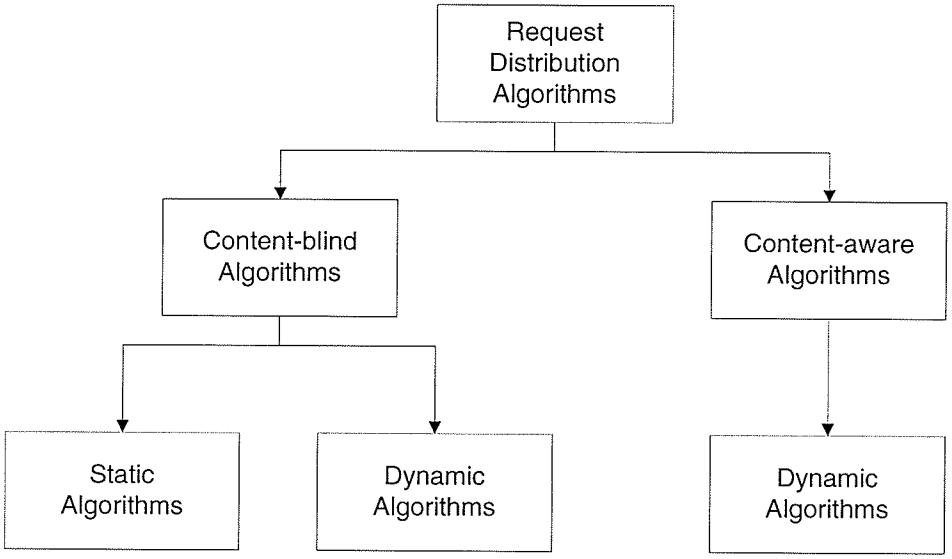

Figure 3.6: Distribution algorithms

Content-blind Static Algorithms

As discussed above, static algorithms do not take into consideration any system state information or knowledge of the requested content while making fowarding decisions. The static distribution algorithms generally use priori system information to make decisions. The major types of static distribution algorithms are the *Round-Robin* algorithm, the *Random* algorithm and a modification of the Round-Robin algorithm called the *static Weighted Round-Robin* algorithm.

The Round-Robin algorithm, also called *cyclic splitting* algorithm uses a circular list to make distribution decisions. In the Round-Robin algorithm the ith request is assigned to the $(i \ mod \ N)th$ back-end node, where N is the number of back-end nodes in the cluster system. Thus, the Round-Robin algorithm just needs to keep a pointer to the last selected server to make distribution decisions. The Random algorithm, also called the *probabilistic* algorithm distributes the incoming request to a back-end node i with

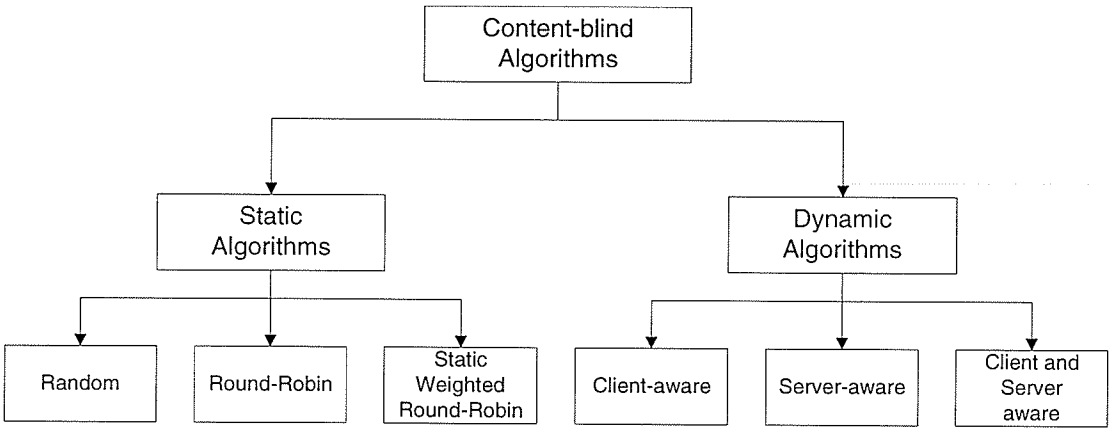

Figure 3.7: Content-blind distribution algorithms

probability p_i, where the probability p_i is selected such that the requests are distributed uniformly through the back-end nodes in the cluster.

In the case where the back-end nodes in the server cluster system have heterogeneous capacities, the assignment policies for the Round-Robin algorithm and the Random algorithm can be slightly modified to take into account the different processing capacities of the back-end nodes. Depending on the processing capacity, each back-end node can be assigned a load factor. The load factor can be assigned depending on the relative back-end node capacity, for example, if J_i is the capacity of the ith back-end node, the relative capacity $\eta_i (0 \leq \eta_i \leq 1)$ is defined as $\eta_i = J_i/max(J)$, where $max(J)$ is the maximum back-end node capacity among all the back-end nodes in the cluster. Now, the Round-Robin policy can use the relative capacity for distribution in the following way. A random number $\sigma (0 \leq \sigma \leq 1)$ is generated, and compared with the relative capacity of the back-end node in the cluster system to send the request to. If the generated random number is less than the relative capacity for that node, we send the request to that node, otherwise we repeat the process for the next back-end node in the circular list. For example, suppose S_i is the next back-end node in the circular list to send the

request to, and η_i is its relative capacity. We send the request to it only if $\sigma \leq \eta_i$, where σ is the generated random number. If not, we move on to the next back-end node in the circular list i.e. S_{i+1} and generate another random number and compare with the relative capacity of S_{i+1}. For the Random policy, the request assignment can be done in the following way. Each back-end node is assigned a different probability depending on its relative capacity, and then the basic Random algorithm is used to distribute the incoming requests.

The static *Weighted Round-Robin* algorithm is a modification of the basic Round-Robin algorithm, and is mainly used to take into consideration the different processing capacities of the back-end nodes in the cluster system. For this, each back-end node in the server cluster is assigned an integer weight ω_i, which is generally calculated as $\omega_i = J_i/min(J)$, where J_i is the capacity of the ith back-end node,and $min(J)$ is the minimum capacity among all the back-end nodes in the cluster. The distribution algorithm can then be set up to distribute the incoming request according to the back-end node weights.

The problem with the static distribution algorithms is that, there are a lot of chances that these algorithms could potentially make poor assignment decisions. Since they distribute the incoming requests according to some hardwired policy, not considering any state information, they can forward requests to back-end nodes which have a serious backlog of requests, while there are back-end nodes which are idle and are thus more suitable candidates for request assignment.

Content-blind Dynamic Algorithms

As discussed above, the dynamic algorithms use some form of system state information to make more intelligent distribution decisions. The content-blind dynamic distribution algorithms mostly work at the TCP/IP layer and hence, do not take into consideration any information about the requested content. Depending on the level of the system

state information used by the content-blind dynamic algorithms, they can be further classified as *Server-aware* algorithms, *Client-aware* algorithms and *Client and Server-aware* algorithms.

The *Server-aware* algorithms distributes requests to the back-end nodes on the basis of some server state information such as load conditions at the back-end nodes, latency in serving the requests, network utilization or availability of the back-end nodes. A key issue in designing Server-aware distribution algorithms is identifying a suitable *load index*. The selected load index should be able to predict the performance of an incoming request if it is to be served by some particular back-end node. The load indexes that have been used include the length of the CPU queue, the average CPU queue length over some period of time, the amount of memory available, the disk or I/O storage utilization, the instantaneous number of active connections, the number of active processes, and the CPU utilization. Depending on the way the load indices are evaluated, they can be classified as *input indices*, *server indices*, and *forward indices*.

The input indices are generally computed at the front-end switch without any co-operation of the back-end nodes. The input indices information is very limited, and uses the number of active connections at the back-end nodes. On the other hand, the server indices are calculated at each server and then transmitted to the front-end node. The server indices can thus provide detailed information about the state of the back-end nodes, such as CPU and disk utilization, number of active connections, and the number of processed packets. Both the input indices and the server indices can be used in unidirectional/one-way architectures and bidirectional/two-way architectures. In unidirectional/one-way cluster architectures, the server indexes are computed by a process monitor running on each of the back-end nodes and periodically transmitted to the front-end switch. In bidirectional/two-way architectures the same procedure can be used or a different procedure can be used where part of the server indices such as the number

of active connections, and the latency time can be calculated at the front-end node and other server indices, such as CPU and disk utilization can be transmitted to the front-end node by the process monitors running at the back-end nodes. The forward indices use information obtained directly by the front-end switch that emulates HTTP requests to the cluster server. The forward indexes are mainly used in unidirectional/one-way cluster architectures only.

Another issue with the server-aware distribution algorithms concers, how and when to compute the load information and how frequently to transmit the information to the front-end switch running the distribution algorithm. The intervals between updates of the load indices need to be evaluated carefully to make sure that the system remains stable. If the load information is transmitted very frequently, it may lead to additional traffic in the server cluster system, and if the load information is transmitted infrequently, the information could become stale depending on the frequency of the updates [MiM97], [DaM00]. In both the cases, this will result in performance degradation even with the use of complex dynamic algorithms.

After selecting the appropriate load index, the front-end switch can use the load index in different distribution algorithms. One simple policy called the *least loaded* policy assigns newly arrived requests to the back-end nodes with the lowest load index. In the *least connection* policy, the front-end node assigns incoming requests to the back-end node with the lowest number of active connections. Similarly, in the *fastest response* policy, the front-end switch assigns incoming requests to the back-end node with the fastest response time i.e., the node which responds fastest to the assigned requests. In another policy, a modification of the static weighted round-robin policy called the *Weighted Round-Robin* policy, each back-end node is assigned a weight that is proportional to its load. This weight is dynamically calculated by the front-end node by gathering load information from the back-end nodes at a regular intervals. The front-end node then uses the assigned

weights to forward incoming requests to the back-end nodes in the cluster.

The *Client-aware* algorithms distribute incoming requests depending on some basic client network information available at the TCP/IP level of the OSI protocol stack. This information could be the the source IP address and the TCP port numbers available in the incoming TCP/IP packets. A simple client-aware policy would be to partition the back-end nodes to run different applications and assign the incoming requests to them depending on the port numbers in the TCP/IP packets or to partition the back-end nodes such that a group of clients could be assigned to them according to their IP addresses.

The *Client and Server-aware* algorithms forward requests based on the client as well as the server state information. These policies combine the client information such as the source IP address and the service port number with some information about the server load. These policies can benefit from client information by assigning consecutive connection requests from the same client to the same back-end node in cases where the initial connection setup process is computationally expensive, for example, as in the case of SSL connections.

3.2.5 Content-aware Algorithms

A front-end node using *content-aware* algorithms works at the application layer (Layer-7 of the OSI protocol stack), and thus has the ability to examine the content requested in the HTTP packets before making any forwarding decisions. It is also able to use additional information regarding the HTTP request, such as cookies. There can be many advantages to using content-aware request distribution algorithms, such as

- increased performance due to improved hit rates in the back-end's main memory caches, thus reducing latency caused in disk accesses.
- increased secondary storage scalability due to the ability to partition the server's

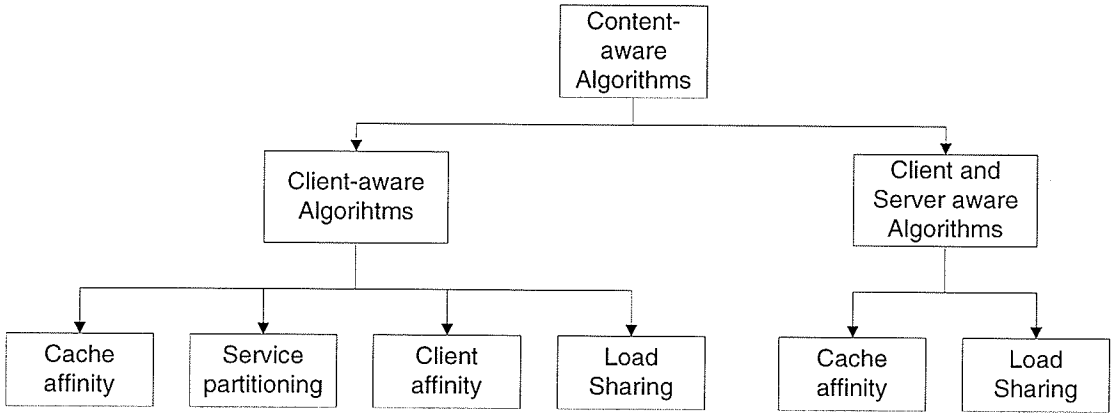

Figure 3.8: Content-aware distribution algorithms

database over the different back-end nodes.

- ability to employ back-end nodes specialized for certain type of requests for e.g., audio, video, and medical images.

The content-aware algorithms can be classified as *Client-aware* algorithms and *Client and server-aware* algorithms as shown in Figure 3.8, which shows the taxonomy of the content-aware distribution algorithms.

Client-aware Algorithms

The *Client-aware* request distribution algorithms can be further classified as shown in Figure 3.8. In the *client-affinity* policy, the front-end node uses client session identifiers such as cookies and SSL identifiers to select a target back-end node. The session information is useful in cases where all the requests from the same client are to be assigned to the same back-end node. For example, in the cases of SSL connections, the initial handshake procedure is a computationally expensive process, as it requires certificate exchanges, encryption and compression negotiation and session id setup. Using the session

identifiers, subsequent SSL sessions can skip the handshake by using the same session ID for some period of time.

The *cache-affinity* policies achieve the highest cache hit rates by statically portioning the working set among the back-end nodes. These policies use a hash function to map the files in the working set to a specific back-end node. The distribution algorithm running on the front-end node uses the same hashing function in selecting a target back-end node for an incoming request. This approach is able to achieve the maximum locality of reference, but still it has several drawbacks. It can only be used for sites providing static content only. It is difficult to balance the load between the back-end nodes, as it is very difficult to partition the working set in such a way that the request distribution is balanced among all the back-end nodes in the cluster.

In the *service partitioning* policy, the back-end nodes are partitioned according to the service type they handle. So, each back-end node can be set up to handle certain type of client requests, such as static content, dynamic content, streaming media files, multimedia files etc. The front-end node is then statically configured to forward the incoming client requests to the respective server handling the content. This approach has the same limitations as the cache-affinity policy as it could lead to some of the servers being overloaded while others are idle if a certain type of content is requested more than the others.

The last policy in the client-aware category is called the *load sharing* policy. The main goal of this policy is to balance the load among all the back-end nodes in the cluster. In one of the load sharing policies used mainly for static content, the front-end node dynamically partitions the working set among all the back-end nodes according to the size distribution. This policy defines a size range associated with each server in such a way that the total load directed to each server is the same [Bac99]. The front-end node selects the target back-end node for serving the request depending on the size of

the requested file.

Client and Server Aware Algorithms

The client aware policies can be easily integrated with some server state information to form client and server aware policies. These policies depend on client information for cache affinity purposes and server state information for load sharing purposes, as shown in Figure 3.8. In one of the policies already discussed in chapter 2 called as Locality-Aware Request Distribution (LARD)[PaA98], the front-end node assigns the incoming requests to the back-end node already caching the document, as long as its load is below some threshold. In the case of overload the request is assigned to the least loaded back-end node. This policy improves load sharing by avoiding overloading at the back-end nodes and improves cache affinity by trying to assign the request to the node already caching the requested object.

Chapter 4

Request Distribution Algorithms

This chapter describes in detail some of the algorithms examined in this study for request distribution in a cluster based server system. The main objective in any request distribution algorithm is to achieve good load balancing at the back-end nodes in the cluster so that none of the nodes are idle while others are overloaded. With the content aware distribution, the aim is to achieve high cache hits in the main memory cache of the back-end nodes in addition to balancing the load. In this study I compare some of the major request distribution algorithms which have been proposed to achieve the above objectives. I compare those algorithms in terms of the throughput achieved (number of bytes served per second between the server and the client), the number of cache hits and misses at the back-end nodes, and how well the load is balanced at the back-end nodes.

4.1 Round-Robin Algorithm

The first is a content-blind algorithm called *Round-Robin request distribution algorithm.* The round-robin algorithm, also called the *cyclic splitting* algorithm has already been explained in chapter 3. This algorithm is very simple in the sense that it does not make

use of any state (client or server) information in making forwarding decisions.

4.2 Greedy Algorithm

The next algorithm considered is called the *Greedy algorithm.* When a connection request arrives at the front-end node, the Greedy algorithm tries to estimate the service time for the request on each of the back-end nodes. For this, the front-end node keeps update information about the total number of requests pending to be served on each of the back-end node's queue, the total size of requests pending to be read from the disk, and the content of the cache memory on each of the back-end nodes. If Greedy finds that the requested document is cached in a particular back-end node, the algorithm estimates the service time on that node by considering the number of requests pending in the back-end node's queue as shown below

$$t = \frac{s(nr+1)}{sbw} \tag{4.1}$$

and if the requested document is not cached on a back-end node i.e., has to be read from the disk, the service time on that node is calculated as follows:

$$t = \frac{tb+s}{dbw} + \frac{s(nr+1)}{sbw} \tag{4.2}$$

Where, s is the size of the requested document, nr is the number of requests in the back-end node's queue, sbw is the back-end node's bandwidth, tb is the total size of the requests pending to be read from the disk, dbw is the disk bandwidth, and t is the estimated service time. After estimating the service time for the requested document on each of the back-end nodes Greedy sends the request to the back-end node with the least service time.

4.3 Locality Aware Request Distribution

The next algorithm I consider is called the*Locality-Aware Request Distribution* (LARD) [PaA98]. This algorithm is a content-aware algorithm, where the front-end switch uses the content requested, in addition to information about the load on the back-end nodes, to choose a target back-end node. This algorithm focuses on achieving both the above said objectives viz. load balancing and high cache hit rates at the back-end nodes. In this, the front-end also limits the number of outstanding requests at the back-end nodes. This allows the front-end to respond to changing load on the back-ends, since waiting requests can be directed to back-ends as capacity becomes available. Figure 4.1 describes the LARD server selection algorithm.

When a request first arrives for a given document, it is assigned a back-end node by choosing a lightly loaded back-end node. Subsequent requests for that document are directed to the assigned back-end node, unless that node is overloaded, in which case, the document is assigned to a new back-end node from the current set of lightly loaded nodes. In LARD, a node's load is measured as the number of active connections on each back-end node.

LARD uses two threshold values for the number of connections on each back-end node. The first is called the lower threshold value (T_{low}) and the second is called the higher threshold value (T_{high}). T_{low} is defined as the load below which a back-end is likely to have idle resources and T_{high} is defined as the load above which a node is likely to cause substantial delay in serving requests. While making a back-end node selection decision for a particular document, if the front-end sees that the load on the node caching the document is greater than T_{high} while another node has a load less than T_{low}. it selects the latter node as the target for the document. Also once a node reaches a load of $2T_{high}$, the request is not sent to it, even if no node has a load less than T_{low}. LARD also limits the

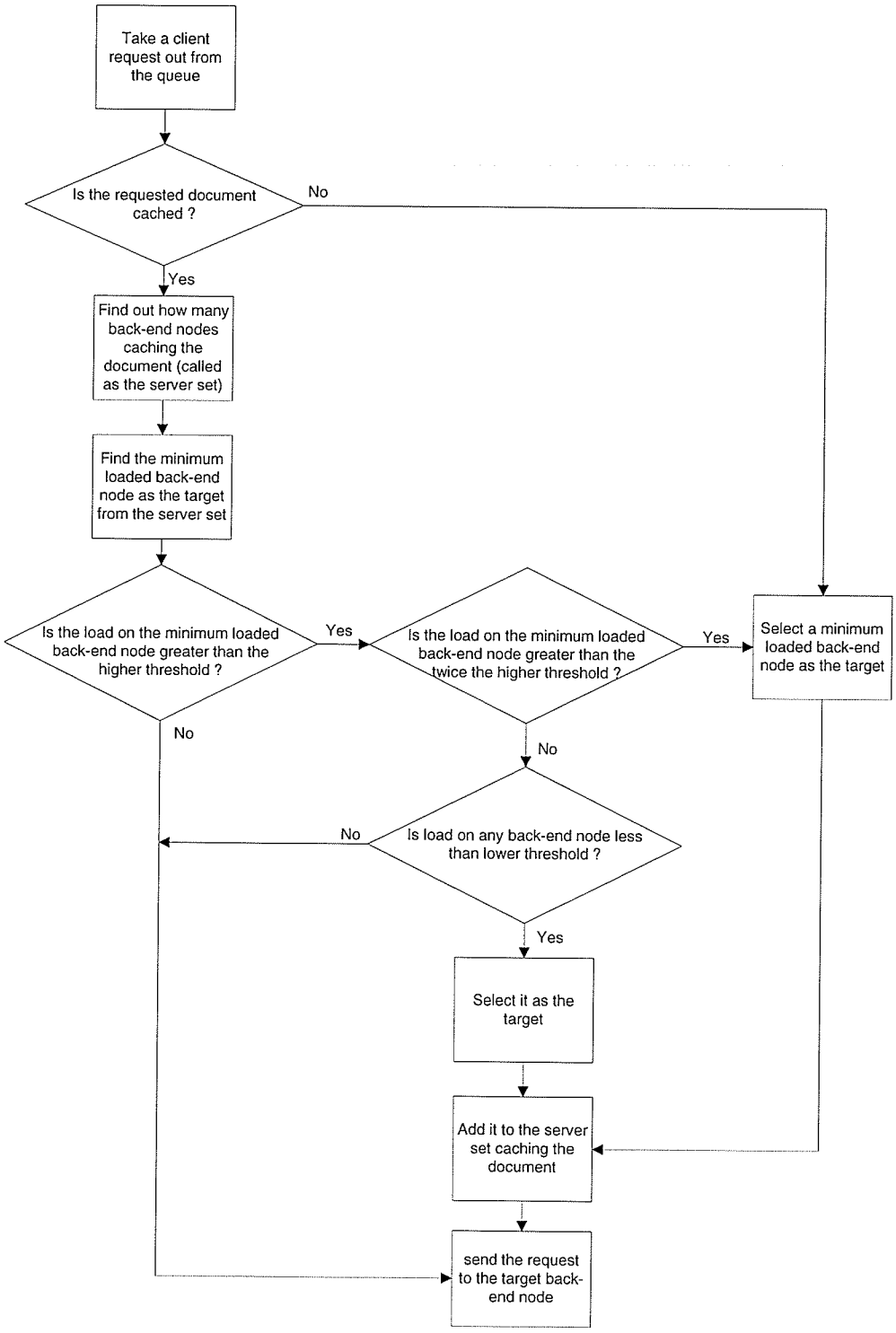

Figure 4.1: LARD server selection algorithm

total number of connections admitted into the cluster to a value of $(n-1)*T_{high}+T_{low}-1$, where n is the number of nodes in the cluster. This is done so that the load on all nodes does not rise to a value of $2T_{high}$.

4.4 The Document Placement Model

The last algorithm considered is called the *document placement model* [DiM01]. The problem here is to determine which back-end node in the cluster system should cache which document, and what percentage of incoming requests for documents should be assigned to which back-end node. The aim here is to represent the problem as a performance optimization problem from the perspective of the clients. To represent the problem, DPM outlines a simple *queuing model*, introduces some approximations, and specifies a nonlinear programming problem with some integer variables. The model used here considers a *best effort service* model.

The *document placement model* is used to place the documents on the back-end nodes and the front-end node is set up to steer the requests to the back-end nodes in the cluster. The front-end switch is assumed to be able to perform content-aware switching, whereby it looks into the content requested before forwarding the request to the target back-end node. The document placement model uses an off-line process as shown in Figure 4.2.

As shown in Figure 4.2 the off-line component of the process uses long-term access statistics to infer some measure of the popularity of the documents being requested. The optimization model is given the measured popularity measurements, the network topology, and the capacities of each of the back-end nodes as input. For the given input parameters, the optimization model is solved to obtain (a) the locations and number of replicas for each document, and (b) the fraction of the demand that is to be satisfied by each replica. After obtaining the solution, the replicas of the documents are placed on

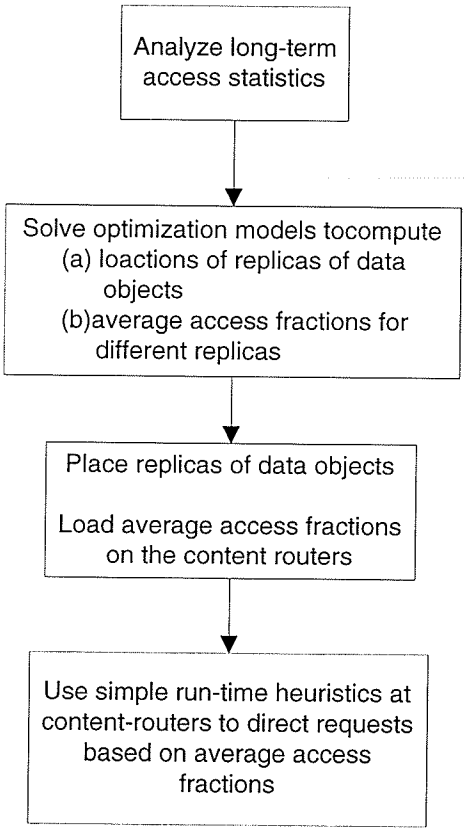

Figure 4.2: The Document Placement Model

the appropriate back-end nodes and the content-aware front-end switche is set up with the fractions of the requests that should be routed to each replica. The content-aware switche will be using some run-time heuristics with the average fraction of requests to decide how the incoming requests should be routed.

4.5 Quality of Service Aware Model

All the algorithms considered so far are based on the *Best effort service* (BES) model, where all the requests are treated equally by the back-end nodes. In this study, I propose

a new Quality of Service (QoS) aware request distribution algorithm. The motivation for this comes from the fact that different clients use different bandwidths to connect to the Internet. The usage of high bandwidth connections like Cable modems and DSL is increasing daily. There is also an increase in the usage of wireless devices like cell phones and PDAs to connect to the Internet. Considering all the clients equally and giving them the same proportion of the server resources leads to a waste of server resources, as the resource demand of a client using a $56K$ modem and a client using a $T1$ line is very different.

In response to this, I have modeled the scheduling algorithm in our server as a round-robin scheduler, which assigns a different proportion of resources to different bandwidth clients. The server is assumed to be providing two classes of service: (a) Quality of Service (QoS) and (b) Best effort service (BES). In my model the clients are divided into different QoS classes depending on their connection bandwidths and some of the clients (usually with the least connection bandwidth) are BES clients. The server resources are shared by the QoS clients with decreasing fraction of resources allocated to clients from highest to the lowest QoS class , and the BES clients are given whatever resources are left over.

To provide differentitated service to different classes, the back-end nodes provide a different fraction of service time to each client request depending upon the class it belongs to. To understand this, let b_j (bytes/sec) be the bandwidth of a client belonging to QoS class i. The request distribution algorithm running at the front-end assigns the request to back-end node k. At the time of assignment there are already some requests pending to be served on that back-end node. Let, O (bytes) be the total outstanding data to be served on that back-end node k, and B_k (Bytes/sec) be the bandwidth of the back-end node k, so the total time T (sec) the node will take to serve the total outstanding data

is given as,

$$T = \frac{O}{B_k} \tag{4.3}$$

Since the scheduler running on the back-end nodes is a round-robin scheduler, it will assign a fraction Δt (sec) of time T to each request in a round-robin fashion until the request is served. For a QoS request the fraction of time Δt is allocated as,

$$\Delta t = \frac{b_j}{B_k} \rho T \tag{4.4}$$

where, ρ is some fraction approximately equal to 10^{-2} to 10^{-3}.

Equation 4.4 shows that clients with higher bandwidth connections will be given higher service time than the clients with lower connection bandwidths.

For a BES request the fraction of time Δt is allocated as,

$$\Delta t = \left(1 - \frac{\sum_j b_j}{B_k}\right) \frac{\rho T}{N} \tag{4.5}$$

where, N are the total number of BES requests on the back-end node k and ρ is some fraction equal to 10^{-2} to 10^{-3}.

Equation 4.5 shows that, the BES clients are offered service time which is not being used by the QoS clients. It also puts a restriction on the total number of QoS clients that are in the system at any moment, in that the total bandwidth requirement of the QoS clients ($\sum_j b_j$) at any moment cannot exceed the bandwidth of the back-end node. This ensures that the QoS clients are always provided the desrired demand. In the case of overload the system converts the incoming QoS requests into BES requests to provide the guaranteed share of resources to the already admitted QoS clients. This is done by the front-end node as explained in the next section.

4.5.1 QARD-LA

The designed request distribution algorithm is a content-based QoS algorithm called Quality of Service aware, locality aware request distribution algorithm (QARD-LA) as shown in Figure 4.3.

In our QoS model I consider that there is a service level agreement for each QoS class. At any moment of time the cluster system can guarantee QoS service to a limited number of clients belonging to the same class. This is called as the connection threshold for that class. If the number of clients for a class exceeds its connection threshold, they may or may not be provided QoS service depending upon the expected arrivals of the other QoS classes. This is done to provide a guaranteed share of resources to each QoS class in case a burst of requests belonging to a single QoS class arrives into the cluster system.

The front-end node maintains a table of connections for each back-end node and updates it whenever a new request is assigned to a back-end node or a request is completed at a back-end node. For this, it does not need to explicitly communicate with the back-end nodes, as all the requests pass through the front-end node and hence it can update the connection table when it assigns a request to a back-end node. When a QoS request comes in, the algorithms checks to see if it can be accommodated into the cluster. This is done by checking the bandwidth already being used by QoS clients on each back-end node. If there is not enough bandwidth on any back-end node to accommodate the QoS request, it is converted to a BES request and sent to the least loaded node in terms of the number of connection on it.

If the request can be accommodated into the cluster system as a QoS request, we need to make sure that the number of connections for that particular class does not exceed the assigned threshold so that the other QoS classes are not deprived of their guaranteed share of resources. But we also need to make sure that the cluster resources are fully

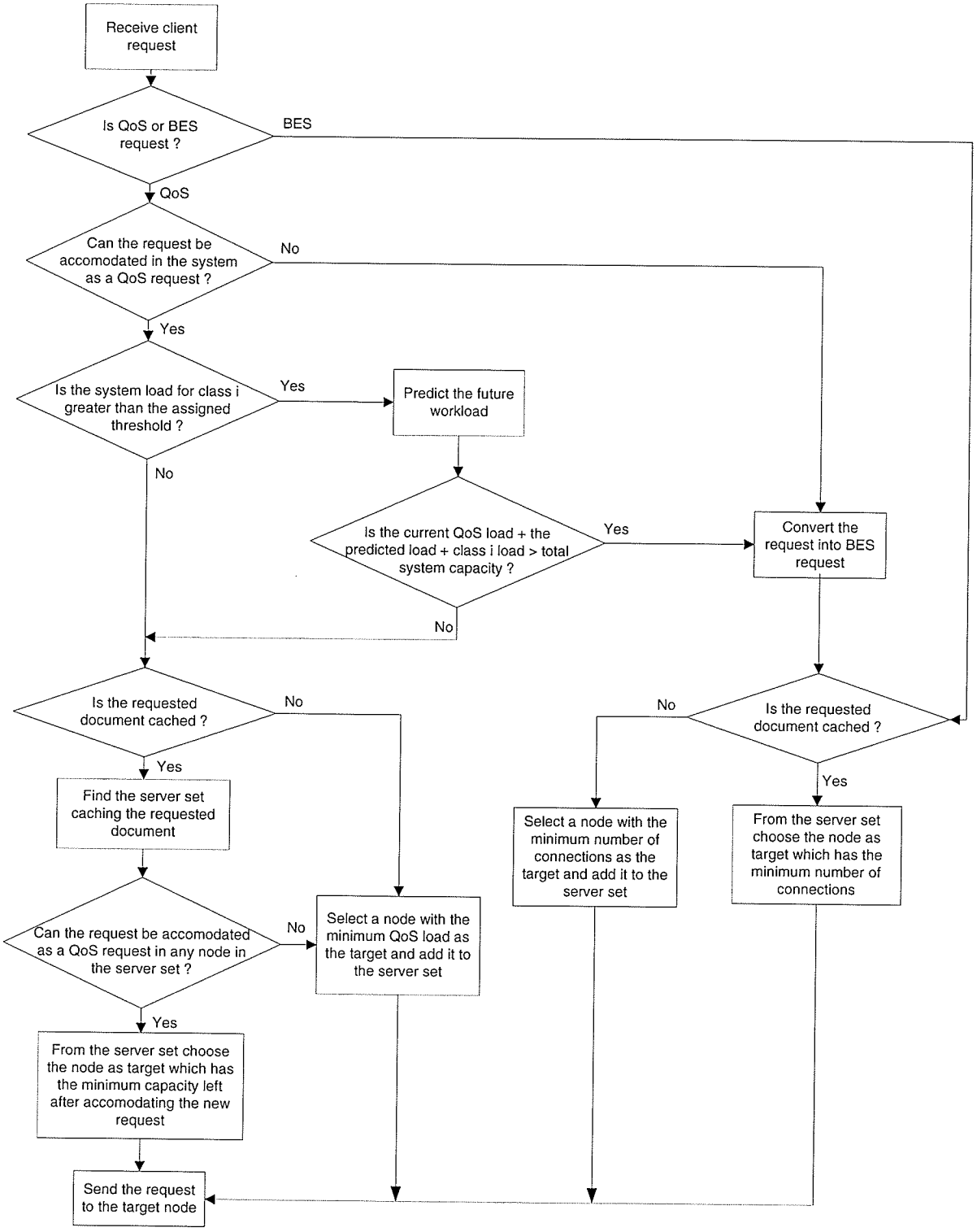

Figure 4.3: QARD-LA

utilized. That is, in case the assigned share is not being used by particular class, it should be free to be used by other classes. We do not want to convert a QoS request into a BES request if some other QoS classes are not fully utilizing their assigned share of resources. In such a case, some requests may be admitted as QoS requests even if the particular class has reached its assigned threshold value. So, in the case a class reaches it's threshold, the front-end node predicts the future load due to all the other QoS classes for the service time of the request (the prediction algorithm is explained later in this chapter). It then calculates the sum of the current system load due to the QoS clients, the predicted load, and the load due to the new QoS request. If the sum exceeds the total capacity of the cluster system (calculated as the sum of the capacities of the individual back-end nodes), the front-end converts the request into a BES request.

Now, the front-end is left with the decision of selecting the most appropriate back-end node for the request (BES or QoS). If the document is being requested the first time, or it is not cached in any node, it is assigned to a back-end node with the least number of connections in the case it is a BES request, or to the back-end node with the minimum QoS client load (the least bandwidth used) in case it a QoS request. The front-end then keeps track of the back-end nodes caching the documents. A set of back-end nodes caching the same document are called as the *server set* for that document.

If the document is cached on the server cluster system, the front-end node treats the QoS request and the BES request differently. If it is a QoS request, the front-end checks to see if there is capacity in the server set for that document to satisfy the required bandwidth demand. If there is, it sends the request to the node which has the minimum bandwidth capacity left after accommodating the request. This is done so that there is enough bandwidth capacity left on the back-end nodes in case a QoS request with higher bandwidth demand comes in. But if the front-end finds that the request cannot be accommodated in any node in the server set for that document, it sends it to a back-

end node with the minimum QoS client load (i.e., the node on which the QoS clients are using the minimum server bandwidth) and adds the back-end node to the server set for that document. In case the request is a BES request, the front-end node sends the request to the back-end node with the minimum number of connections from the server set for that document.

4.5.2 QARD-LB

In order to make sure that the designed request distribution was efficient, I modified QARD-LA to make a content-blind QoS aware algorithmi, which much like the greedy algorithm sends the request to the least loaded back-end node. I call this algorithm as load balancing, quality of service aware request distribution algorithm (QARD-LB) as shown in Figure 4.4. The initial part of the algorithm where the decision is made to send a client request as a QoS request or to convert it into a BES request depending upon the current load and the number of connections for that QoS is same as in QARD-LA. The back-end node selection mechanism is different for both. In QARD-LB, the front-end node sends a QoS request to the back-end node with the least QoS client load (i.e. the node on which the QoS clients are using the minimum server bandwidth), and the BES request to the node with the minimum number of client connections.

4.6 Predicting the Request Arrival

As explained earlier, the QARD-LA and QARD-LB require prediction of request arrivals for each class in order to decide whether to provide QoS to a request from a class which is already using all of it's assigned resource share. To predict the request arrivals for each class, I model the process as an AR(1) process [ChG02] (autoregressive of order 1). This is a simple linear regression model in which a sample value is predicted based on

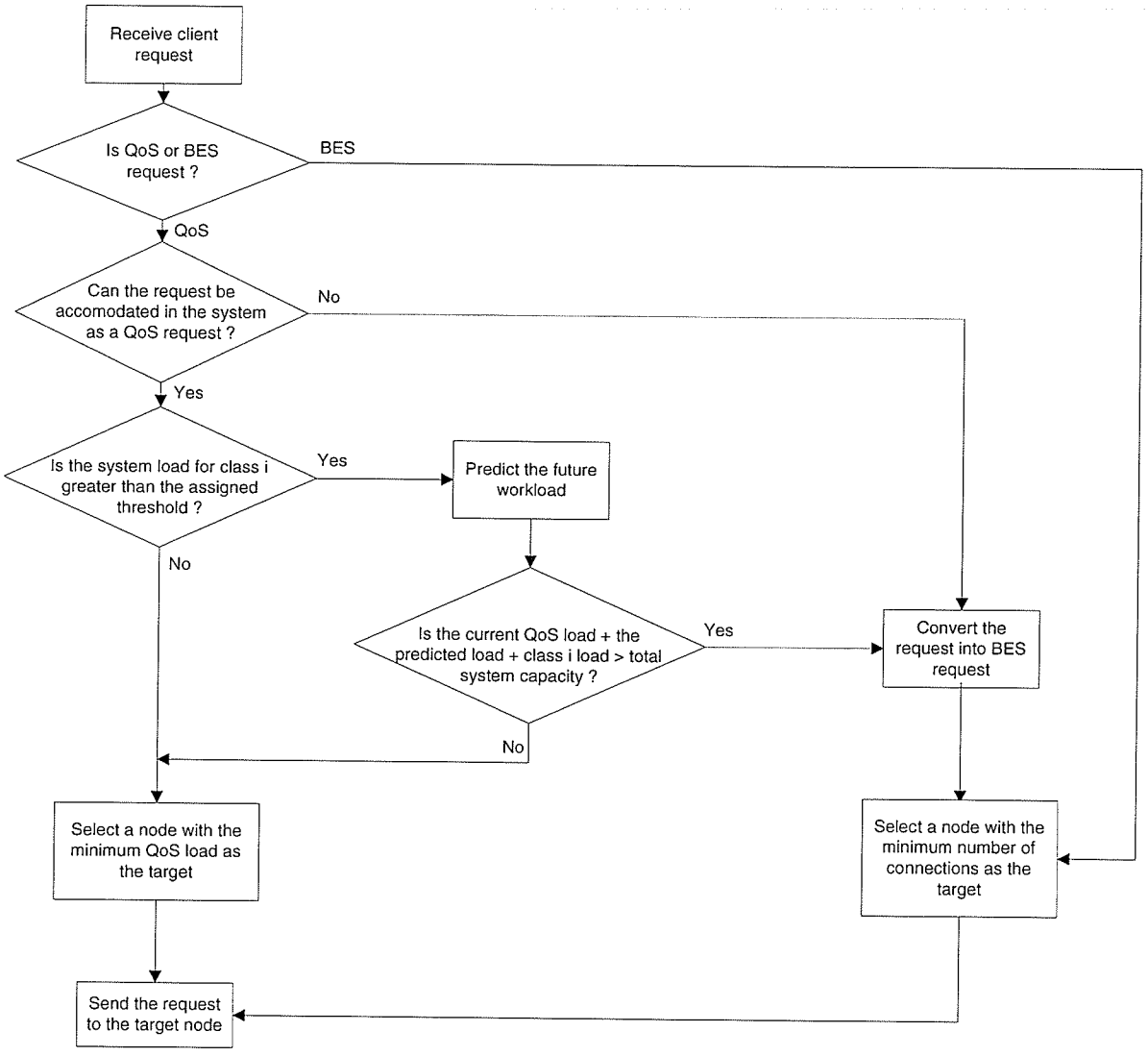

Figure 4.4: QARD-LB

the previous sample values. To do this, the monitoring module measures the number of request arrivals a_i for each class in each measurement interval of duration I. The prediction algorithm then tries to predict the number of arrivals n_i for the next interval. In this case the interval is assumed to be the mean service time for a request. I divide the measurement interval for each class into N smaller intervals called windows (W_i^j, $j = 1$ to N). So the number of requests in each window $\left(a_i^1, ..., a_i^N\right)$ added together give the total number of requests in the measurement interval I ($A_i = \sum_{j=1}^{N} a_i^j$). The monitoring module keeps a history H of the number of requests from each class by updating the numbers in the windows. The prediction module then uses the history H to predict the number of requests for each class according to

$$\hat{a}_i^{j+1} = \overline{a}_i + R_i(1).(a_i^j - \overline{a}_i) + e_i^j, \tag{4.6}$$

where, R_i and $\overline{a}_i$ are the autocorrelation and the mean of the values from the history for each class respectively, and e_i^j is a white noise component. I assume e_i^j to be 0, and a_i^j to be estimated values $\hat{a}_i^j$ for $j \leq N + 1$. The autocorrelation $R_i(l)$ is defined as,

$$R_i(l) = \frac{E[(a_i^j - \overline{a}_i).(a_i^{j+l} - \overline{a}_i)]}{\sigma_{a_i}^2}, 0 \leq l \leq N - 1, \tag{4.7}$$

where, σ_{a_i} is the standard deviation of the values from history for each class and l is the lag between sample values for which the autocorrelation is computed. Thus, using AR(1) model I compute $\hat{a}_i^{N+1}, ..., \hat{a}_i^{N+N}$, where $\hat{a}_i^j$ denotes the estimated value of a_i for the j_{th} window in the interval of length $\frac{I}{N}$. Here I assume that the measurement interval and the prediction interval are the same and also that the history H is equal to the measurement interval I. Then the estimated number of arrivals $\hat{n}_i$ for the prediction interval is given by

$$\hat{n}_i = \sum_{j=N+1}^{N+N} \hat{a}_i^j. \tag{4.8}$$

Chapter 5

Simulations

5.1 Simulation Setup

This chapter describes the design of the simulator and discusses the results obtained using it. The simulator models a server architecture which contains more than one node for serving the client requests, called a Cluster based server architecture. My simulator can be divided into two units. The first unit simulates a front-end for the cluster architecture which is responsible for receiving all the client requests and making decisions as to which node will be serving the given request. It is in the first unit where the different request distribution algorithms will run. The second unit models a typical round-robin processor sharing server, which receives the requests assigned to it by the front end and serves the request. The second unit is more or less the same for all the techniques I am looking into with the exception of the greedy method which requires some modification to the second unit(server) as well. From here on I will call the first unit the front-end and the second unit the server.

5.1.1 The Front-end node

The front-end is set up to model different request distribution algorithms. The details for each are described in the following sections:

Round Robin

For the Round Robin request distribution algorithm the front-end simulator models a simple cycle splitting algorithm, where it forward the ith request to the $imodN$ back-end node, where N is the number of back-end nodes in the cluster server. This is the simplest algorithm as it a content-blind algorithm and also does not have to keep track of any server state information whatsoever.

LARD

The LARD algorithm is a content-aware algorithm, and needs the information about the load on the back-end nodes as well as the cache memory information for each of the back-end nodes. The simulator is set up as shown in Figure (4.1).

Greedy

As the name suggests, the Greedy algorithm tries to select that back-end node as the target which will provide the least service time for the client request, or in other words it selects the least loaded back-end node in terms of the actual size of requests pending on the back-end node. Greedy has been explained in chapter 3. For the simulation, the front-end node keeps track of the actual size of requests pending on each back-end node as well as the content of the cache memory. This load information is updated continuously by the back-end nodes by informing the front-end if there is any change in the load on them. This happens in the following cases: when a requested document has

to be read from the disk, when the disk finishes writing the requested document into the memory, and when the back-end node completes serving a client request. Another case where the load on the back-end node changes is when a request arrives at the back-end node. But the back-ends do not need to inform the front-end node about it, as it is the front-end which forwards the requests to the back-end nodes and hence can update the load information for the target back-end node when it sends the client request to it.

The Document Placement Model

The simulator for the Document Placement Model uses the optimization model as explained in Chapter 4 to solve for the locations and number of replicas for each document and the fraction of the demand that is to be satisfied by each replica. The optimization model also generates information about which document would be cached on which node. The front-end node is then set up with this information. Whenever a client request comes in, the front-end first checks to see which replica would be serving that request, and then it checks to see if that replica is allowed to cache that document or not. If that replica is not allowed to cache the particular document, it reads that document from the disk and serves it without putting it into the memory cache.

5.1.2 The Server Entity

The server entity receives the requests forwarded to it by the front-end and enqueues them in a queue. The server uses a scheduling algorithm, which determines which request is to be processed next. For my case the scheduling algorithm works in a Round Robin fashion. When a given request is pulled out from the queue, the server checks to see if the request is already in its memory cache or not. If the requested document is not found in the memory cache, the server starts reading the data from the disk. In my simulation

the disk entity also works in a round-robin fashion. When a document is to be read from the disk, it is enqueued in the disk queue. The disk then pulls each request out of the queue and serves it for sometime and then returns it to the queue. But, if the request has already been read from the disk or is found in the memory cache, the server entity starts processing the request by assigning it some processing time. The interval of time during which a request is permitted to remain in service is based upon the Best effort service model. During the assigned processing time the server writes data into a window. The window can be considered as a buffer into which the server is writing the data and from which the client is reading the requested data. After writing into the window the server checks to see if it has completed the request during the assigned time, and if not, the request is inserted back into the server queue until the scheduling algorithm takes it out of the queue to be processed again. The scheduling algorithm then goes to process the next request in the queue. The simulation also takes care not to send requests for the same document from some other client to the disk while it being read from the disk. If the server does not take care of it, then the disk wastes time writing the same data twice into the memory.

In my simulation, I have modeled the server to take care of the very slow clients which are reading data from the window at a very slow rate. If the server processes a request from the slow client and sees that it has filled up the window up to its maximum capacity, it will just ignore that request until the client has read about half the window data. After that the server will start processing the request in the usual way. If this is not implemented then the server will keep on processing the request from the slow client again and again writing small chunks of data into the window thus delaying the processing of requests from potential clients who are able to read data at a much higher rate. The cache replacement policy I used for all the simulations is the Least-Recently Used (LRU) policy. In the LRU policy, the least recently used element is evicted from the cache to

Number of requests	Total bytes requested	Working set size	average file size (bytes)
500,000	2065058075	43950206	4130

Table 5.1: Workload-1 Statistics

make room for more frequently accessed documents. The server entity is common for all the distribution algorithms except for the *greedy* where it has to continuously inform the front end about its updated load information.

For QARD-LA and QARD-LB the server entity is almost same as explained above with the only difference that the scheduling algorithm differentiates between the different classes of QoS requests and the BES request by assigning them different share of the processing time as explained in Chapter 4.

5.2 Results and Discussion

This section describes the results for comparison of the different BES request distribution algorithms, and the designed QoS aware request distribution algorithms.

5.2.1 The BES Algorithms

To compare the different request distribution algorithms I used access logs from requests made to the 1998 World Cup Web site. The tracefile I have used consists of 500,000 requests. Table 5.1 gives the total bytes requested and the total working set size of the tracefile. For the simulation, I used the first 50,000 requests for warmup, and then start collecting the results.

I have used the following metrics to compare the different request distribution algorithms:

- Throughput - This is given as the number of bytes transfered per second between the back-end nodes and the clients. It is calculated as the total bytes transfered divided by the sum of latencies for all requests.

- Cache hit ratio - This is the number of requests that are served from a back-end's main memory cache divided by the number of requests in the tracefile.

- Average load imbalance - This is the average of the difference between the load on the maximum loaded and the minimum loaded back-end node at any instant of time.

- Total load imbalance - This is the difference of the number of requests served between the back-end nodes which served the maximum number of requests and the least number of requests.

Figure (5.1), (5.2), (5.3), and (5.4) show the plot of throughput, cache hit ratio, average load imbalance, and the total load imbalance respectively as a function of different sizes of the main memory cache on each back-end node for each of the request distribution schemes.

As can be seen from Figure (5.1), the Greedy request distribution scheme has the highest throughput for all values of the main memory cache size. This can be attributed to the upto date and accurate information about the actual loads (in terms of the actual size of pending requests) on each of the back-end nodes that greedy uses to make forwarding decisions. The throughput achieved by LARD is also equal to that of Greedy where the total main memory cache size of the cluster is less than 0.5% of the total working set. LARD uses the number of connections as the measurement for load, which is not a very effective measure of load, but still achieves throughput close to Greedy. This is because it uses the memory efficiently by achieving a much higher cache hit ratio as shown in

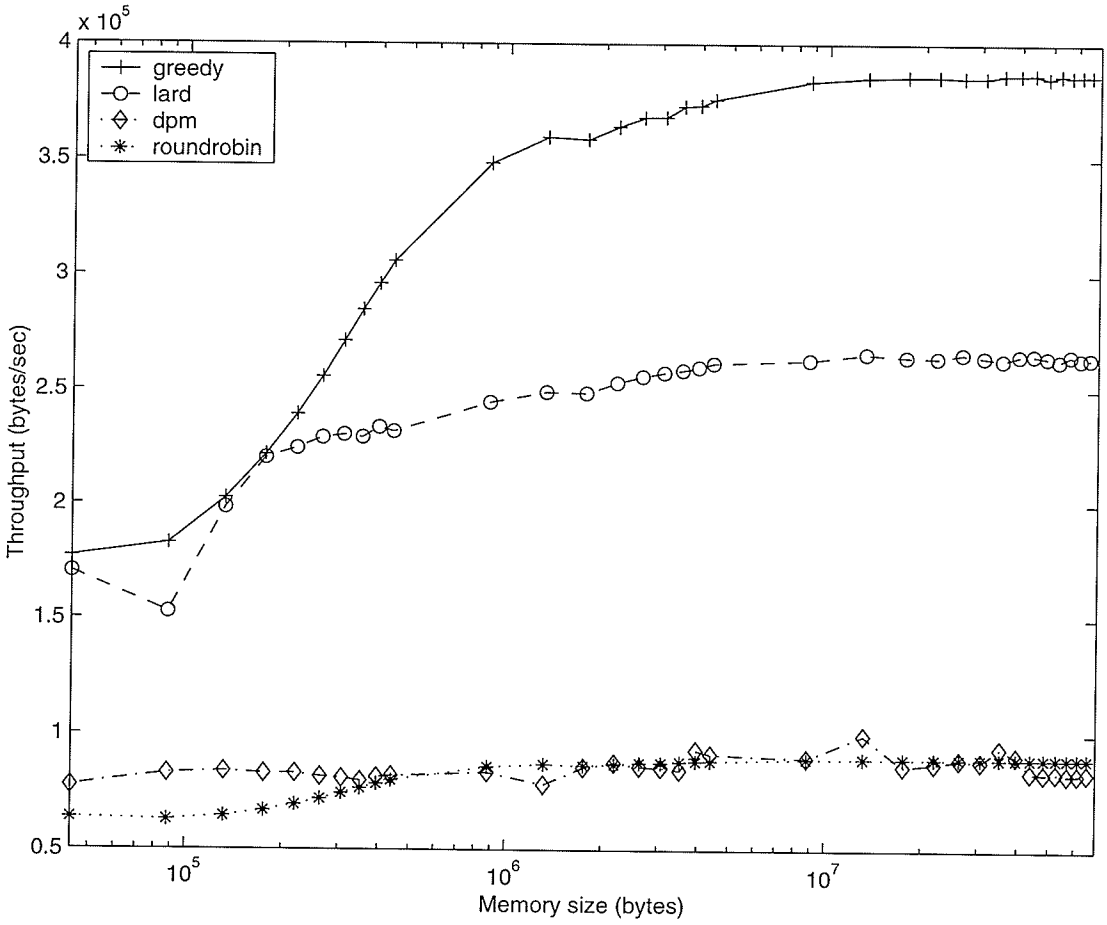

Figure 5.1: Throughput (bytes/sec)

Figure (5.2), thus reducing the overhead caused in the disk access.

Both the Document Placement Model (DPM) and the Round-Robin schemes achieve a very low throughput for all values of memory sizes as compared to LARD and Greedy. Round-Robin achieves the lowest hit ratio resulting in frequent disk accesses. It also has the highest average load imbalance resulting in some of the nodes being overloaded while others being idle. However, the total load imbalance is lowest for the Round-Robin scheme as it sends an equal number of connections to all the back-end nodes. The DPM uses the locality information about the requested documents but still achieves a very

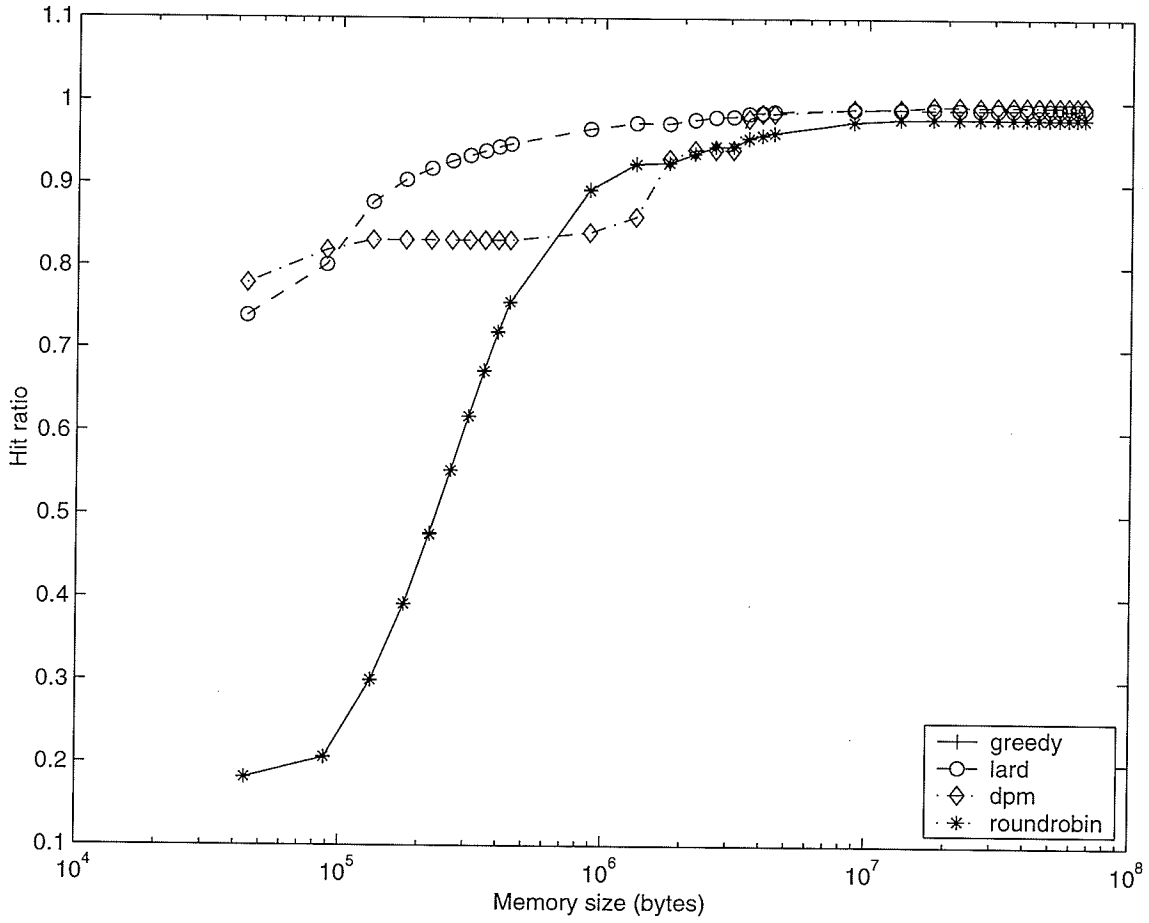

Figure 5.2: Hit ratio

low throughput. Due to the locality information used, DPM achieves a hit ratio almost equal to or slightly lower than LARD for all values of memory cache size, but the average load imbalance achieved by DPM is worse than Round-Robin. The total load imbalance for DPM is also worse than all the other schemes. This shows that the DPM reduces the disk overhead by using the locality information, but at the same time, it results in forwarding most of the requests to overloaded nodes resulting in greater latency in serving the requests.

The hit rate plot shows that the content-aware algorithms LARD and DPM achieve

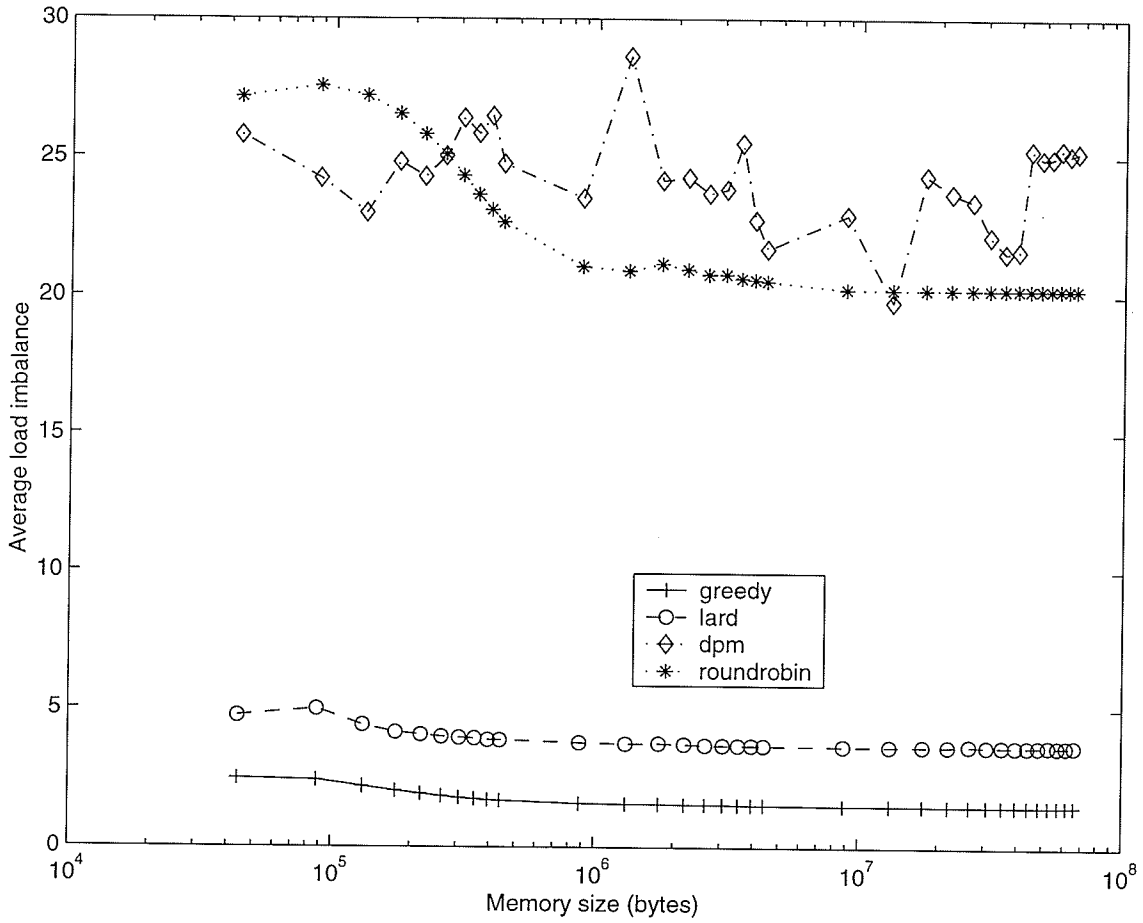

Figure 5.3: Average load imbalance

higher hit ratio than the content-blind algorithms for lower values of the memory size. But as the memory size is increased, all the algorithms achieve almost the same hit ratio. This happens when the memory on each of the back-end nodes is equal to about 0.2% of the total working set. This suggests that to achieve a high cache hit ratio, the memory size on each of the back-end nodes need not be equal to the total working set. This can be achieved at a much lower memory size even if a content-blind request distribution algorithm is used. This also shows that most of the requests are for a small fraction of objects from the working set thus resulting in a very high hit rate even when the memory

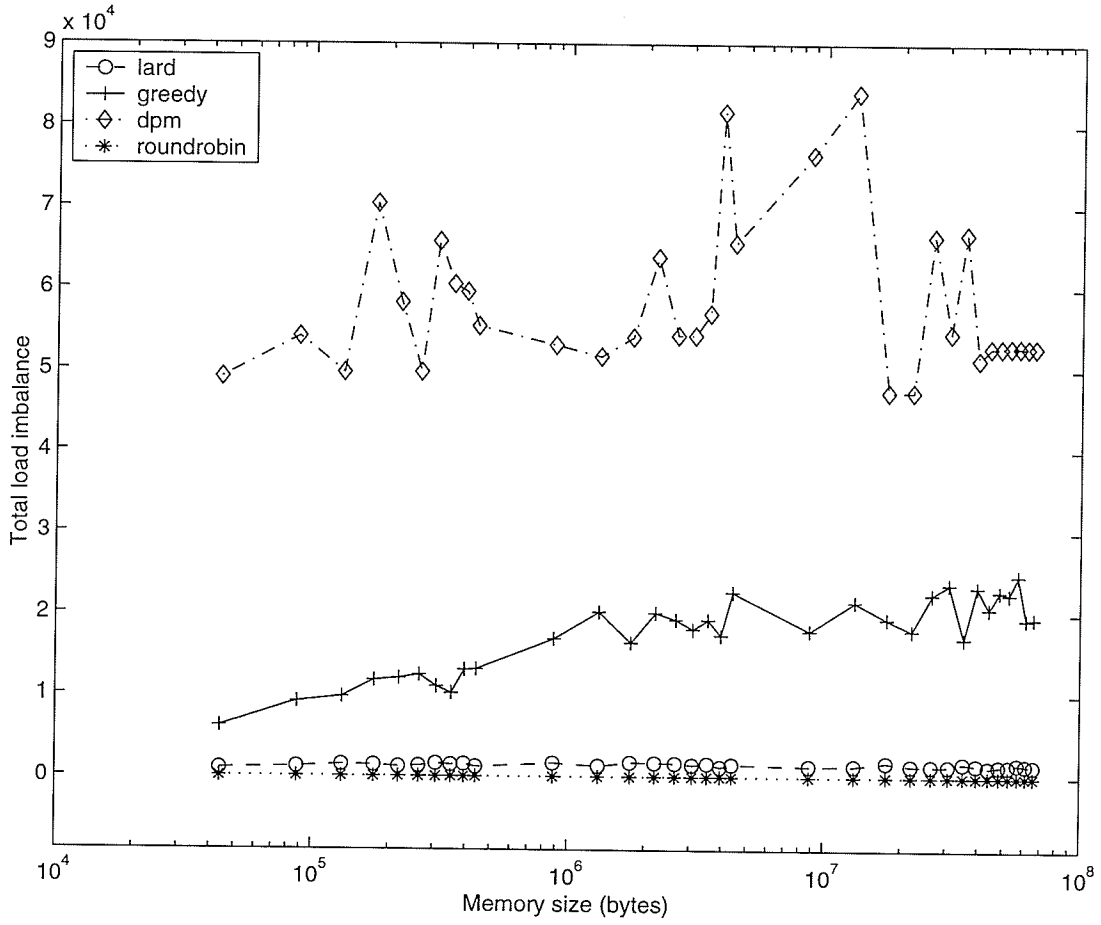

Figure 5.4: Total load imbalance

size is equal to a small fraction of the total working set.

The plot of the average load imbalance and the total load imbalance suggest that having the upto date and accurate information about the actual size of requests pending on each of the back-end nodes as used by Greedy results in better load balancing than using the number of connections on the back-end nodes as the load measure as used by LARD. But the difference between the two load measures is not very significant. But having the most current and up to date information about the actual size of the requests pending on the back-end nodes is not practical and will result in a lot of overhead

Number of requests	Total bytes requested	Working set size	average file size (bytes)
100,000	423888926	24074897	4238

Table 5.2: Workload-2 Statistics

and traffic between the back-end nodes and the front-end node. Using the number of connection as the load measure does not lead to any overhead or traffic since, the front-end can update the information while making server selection decisions, thus making it a more practical and effective measure.

5.2.2 The QARD Algorithms

For this case I used a similar trace file, with the access logs from the requests made to the 1998 World Cup used for comparing all the BES algorithms. The tracefile consisted of 100,000 requests. Table 5.2 gives the important statistics for the tracfile. For this simulation, the first 20,000 requests are used form warmup, and then the simulation begins gathering the statistics.

The simulation uses 9 classes of QoS clients with different bandwidth requirements and a BES class. The connection threshold for each QoS class is set to 10. I also assume that there is some revenue generated by serving each class. The revenue is based upon the bandwidth requirement of a QoS class i.e., the higher the bandwidth requirement the higher the revenue. Table 5.3 shows the bandwidth requirements and the connection threshold values for each class. The total cluster server bandwidth is set such that it's bandwidth is fully utilized when there are 10 connection requests for each of the QoS classes. Using the values shown in Table 5.3, the server bandwidth is set to a value of $912500 KBytes/sec$.

QARD-LA and QARD-LB use a prediction module as explained in Chapter 4. The

class	1	2	3	4	5	6	7	8	9	BES
Bandwidths (Kbits/sec)	2000	1000	900	800	700	600	500	400	300	100
threshold	10	10	10	10	10	10	10	10	10	

Table 5.3: Bandwidths and threshold values for the QoS classes and the BES class

class	Same % requests		Different % requests	
	Actual numbers	Predicted numbers	Actual numbers	Predicted numbers
1	623	590	2183	2147
3	636	707	314	247
6	682	722	292	251
8	843	747	958	922

Table 5.4: Actual and Predicted number of requests for different classes

prediction module needs to know the service time for each request in order to make decisions. But the prediction module may not have prior knowledge of the service time for each individual service access. I assume that the prediction module has knowledge of the service time of the average file sized request. This can easily be achieved through monitoring the access patterns, or through service provider's specifications. In order to verify the prediction module, I tried to predict the number of requests for each class using some tracefiles. The first tracefile used had the same percentage of requests for each class. I also ran the prediction module on a tracefile with a different percentage of requests for each class. Table 5.4 shows the actual and the predicted values for some of the classes with a 5 second interval for both cases. It can be seen that the values produced by the prediction module are a good estimate of the actual numbers.

I compare QARD-LA and QARD-LB using the metrics explained in the last section.

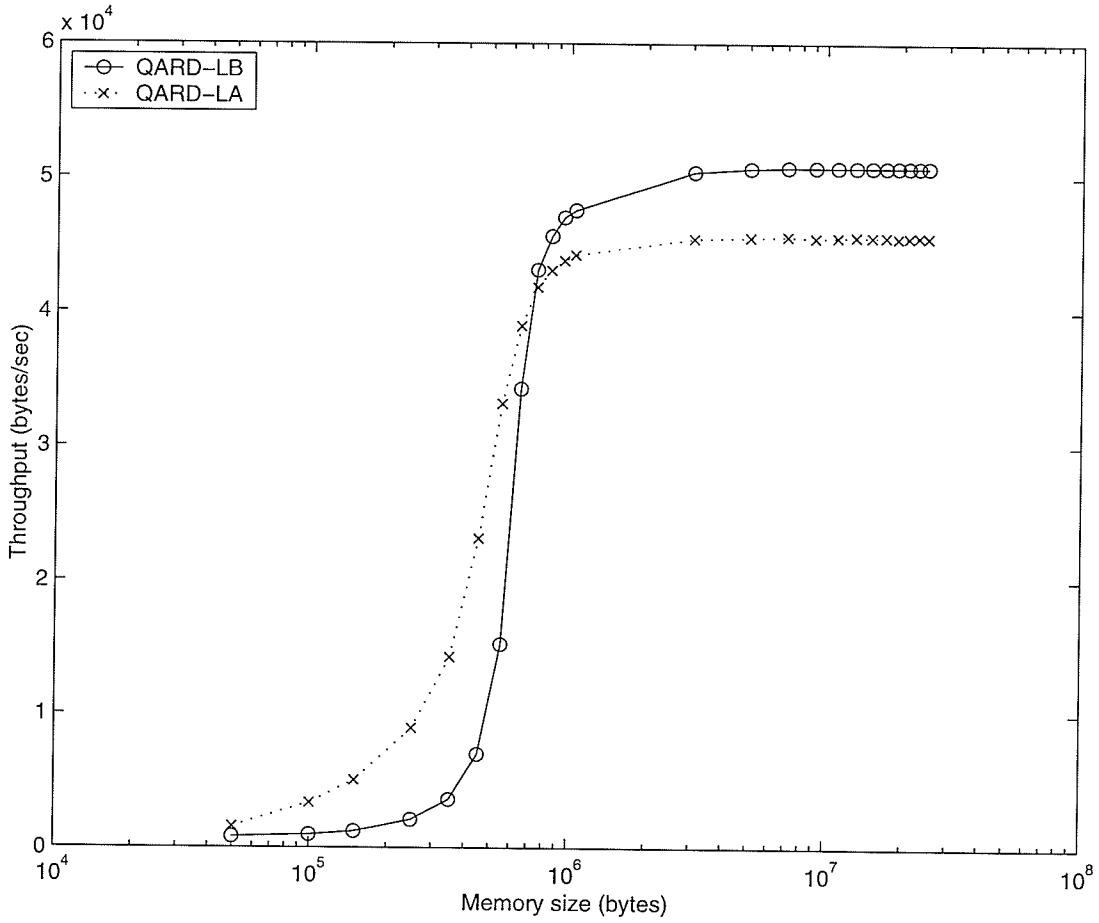

Figure 5.5: Throughput (bytes/sec)

I also use two new metrics as explained below:

- Revenue Generated - This is the total revenue generated in serving a QoS request according to the required demand.
- Discontent Generated - This gives the total number of QoS requests that could not be provided with the desired demand i.e. were served as BES requests.

Figure 5.5, 5.6, and 5.7 show the achieved throughput (bytes/sec), the hit ratio, and the total load imbalance, respectively for QARD-LA and QARD-LB as a function of

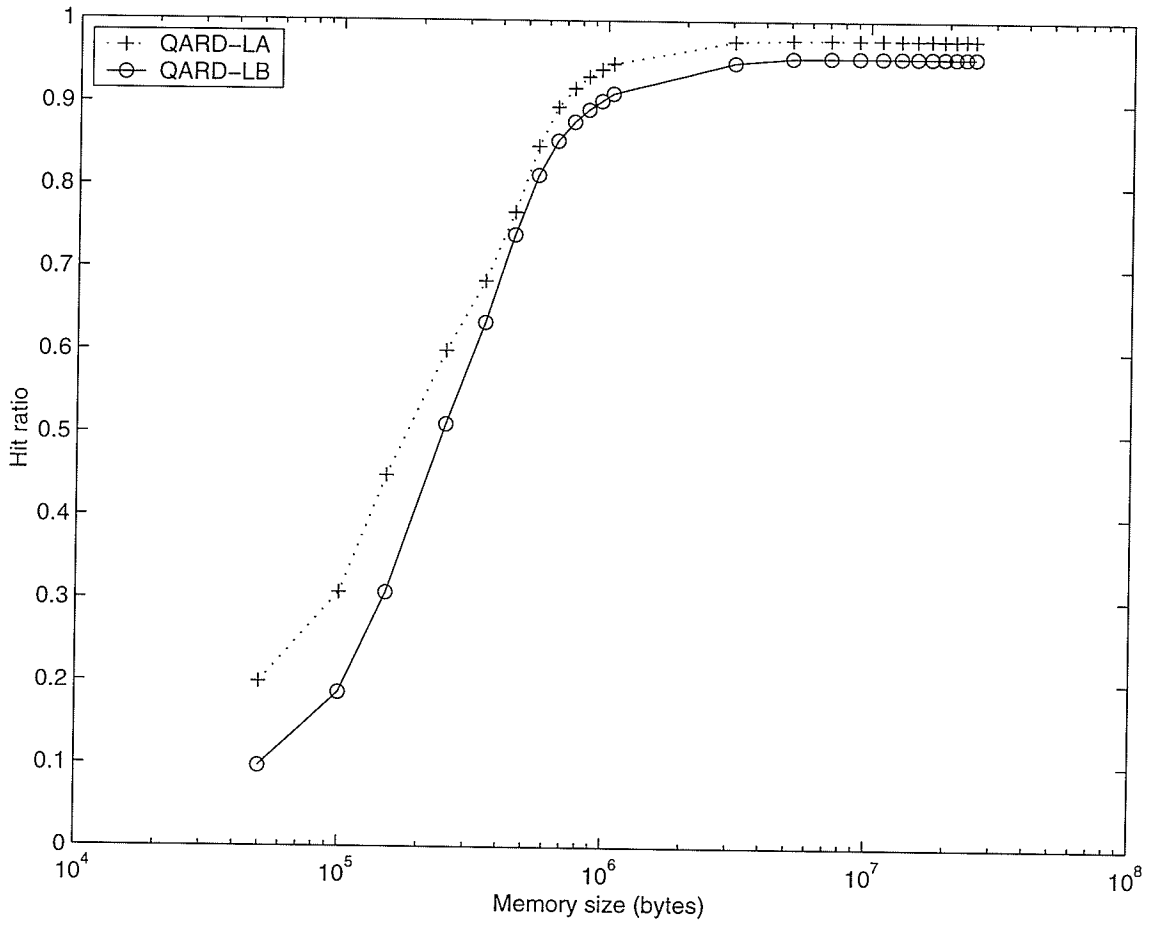

Figure 5.6: Hit ratio

the size of the main memory cache on each of the back-end nodes. The difference in performance of QARD-LA and QARD-LB for the achieved throughput, hit ratio and average load imbalance is due to the same reasons as explained in the last section. As can be seen from the plot of throughput, QARD-LA achieves a higher throughput than QARD-LB for smaller values of the memory sizes (less than 0.2% of the total working set size). QARD-LA uses locality information in addition to the load information to make forwarding decisions, but QARD-LB only uses the load information. Thus, QARD-LA is able to reduce the overhead caused in the disk access at very low values of the memory

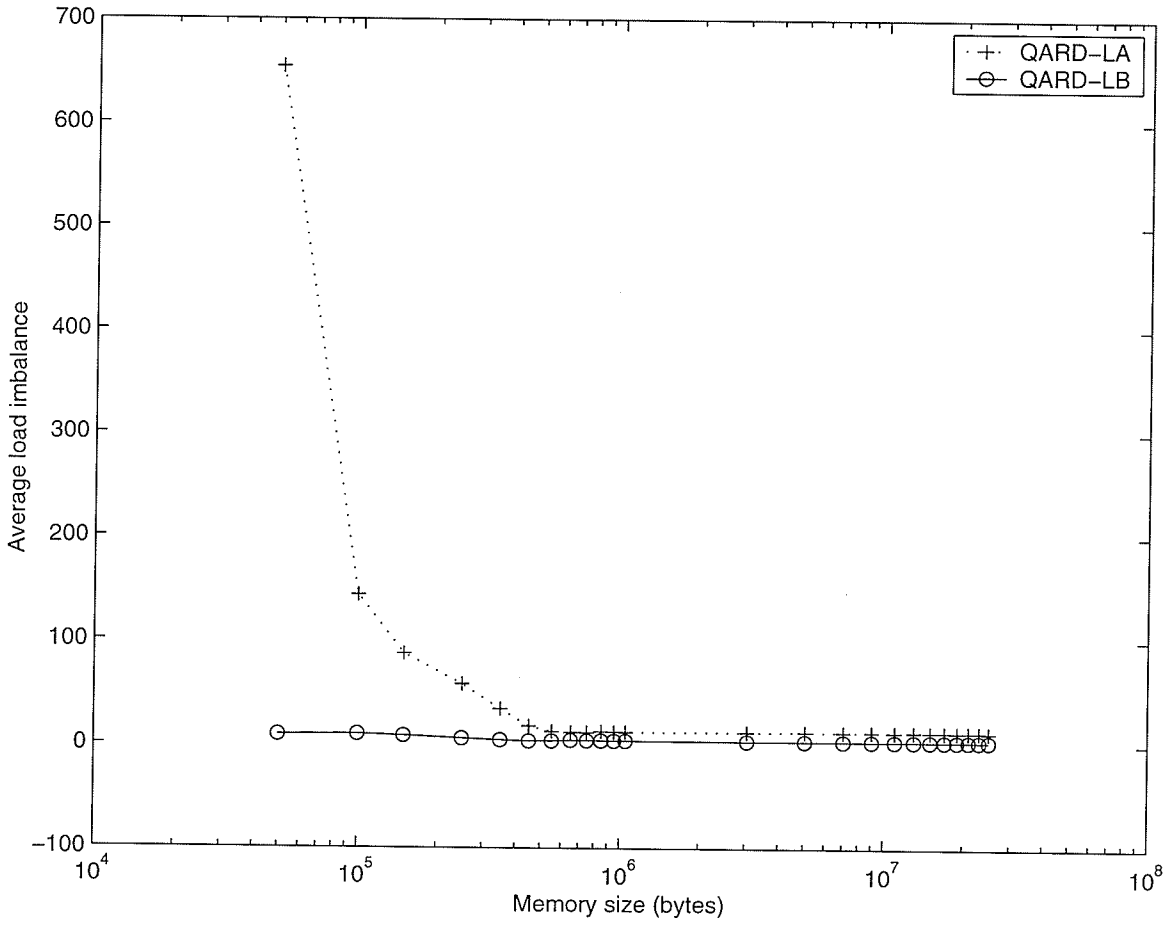

Figure 5.7: Average load imbalance

sizes which is not possible with QARD-LB. From the plot of average load imbalance we note that, for memory sizes less than 0.02% of the working set on each of the back-end nodes the load imbalance achieved by QARD-LB is much better than that achieved by QARD-LA, but still it is not able to achieve the same throughput. Thus, it can be said that at very low values of memory size the locality information, with a little knowledge about the load, can give a much better performance than that achieved by just load balancing alone.

For a QoS aware request distribution algorithm the aim is to increase the generated

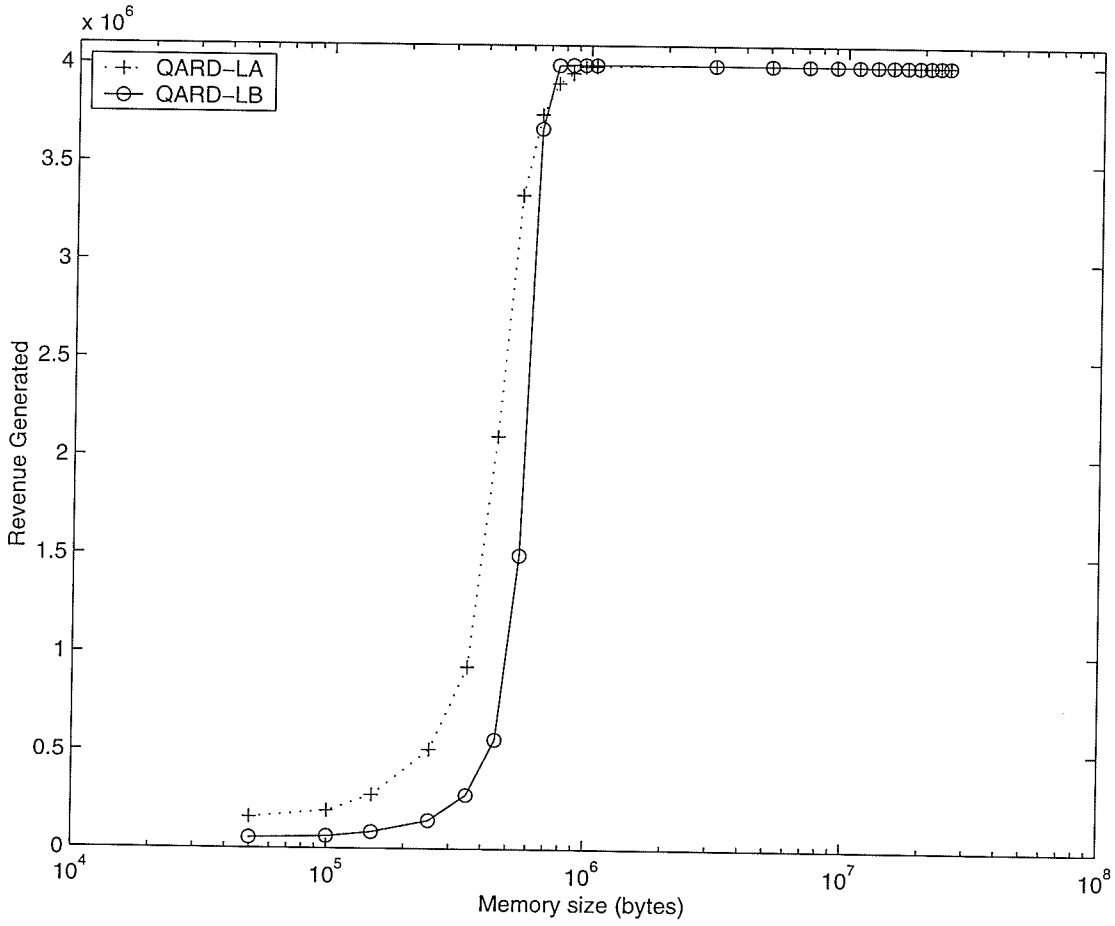

Figure 5.8: Total Revenue Generated

revenue and decrease the discontent by serving requests at their desired demands. Figure 5.8 and 5.9 show the total revenue generated and the total discontent generated respectively for QARD-LA and QARD-LB as a function of varying main memory sizes. QARD-LA is able to generate a higher revenue than QARD-LB. This is because using the load information and locality information QARD-LA is able to accommodate the QoS requests more efficiently than QARD-LB on the cluster QARD-LA is able to serve requests at a higher rate than QARD-LB for smaller values of memory thus making room for other requests and resulting in an increase in the revenue generated.

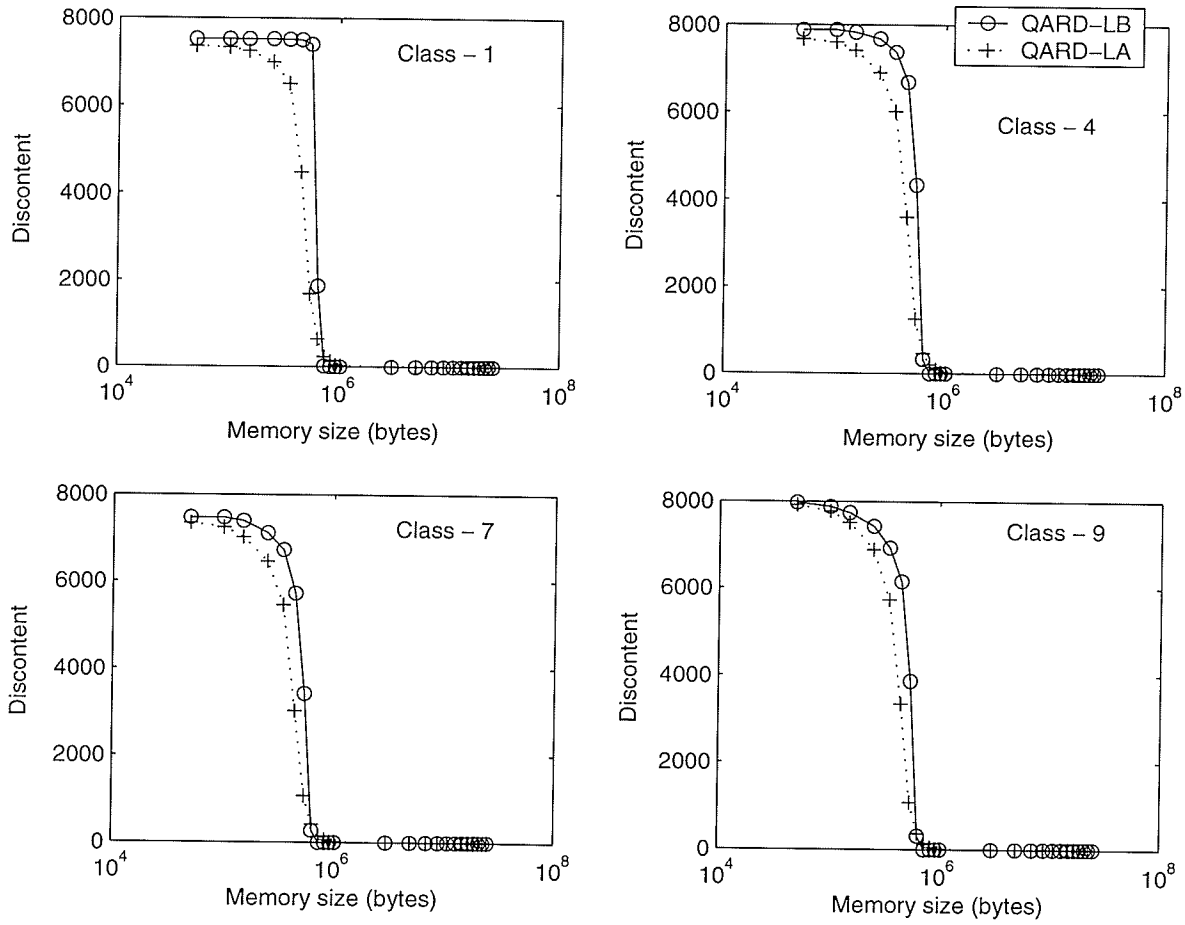

Figure 5.9: Total Discontent

The plot of discontent shows the discontent generated for some of the QoS classes. It can be seen that for memory size less than 0.02% of the total working set on each cluster, QARD-LA is able to serve more requests with their desired demand than QARD-LB. Hence, it is able to reduce the generated discontent and increase the total revenue generated.

For memory sizes greater than 0.02% of the total working set on each of the back-end nodes, both QARD-LA and QARD-LB are able to achieve similar performance. This suggests that for larger values of memory size only a simple load balancing strategy can

be sufficient to achieve best results. But in reality, it is impossible or not very cost effective to have memory sizes closer to the size of the working set.

Chapter 6

Conclusions

In this thesis I present a QoS aware request distribution scheme for Web server clusters. In addition to providing differentiated support to different classes of clients, this scheme tries to maximize the achieved throughput by efficiently using the cluster's main memory cache and by trying to achieve good load balancing between the nodes. The main reason for developing a new request distribution technique is that current web server clusters lack support for providing QoS to their clients. Most of the work on clusters focus on how to maximize the utilization of resources of the cluster. Some of the work done on QoS support for web server clusters tries to allocate the resources on the cluster between different classes depending upon the request arrival rates for that class, but none of the work tries to combine the request distribution with the resource allocation. The reason for doing this was to guarantee the allocation of resource share for each class of clients. At the same time, the scheme will try to efficiently use the main memory cache and balance the load between the nodes.

In order to find out which factors effect the performance (memory usage, load balancing, throughput) in a cluster design, I compared some of the major request distribution algorithms currently being used. These included content-blind as well as content-aware

request distribution algorithms. I found that a simple load balancing scheme can achieve throughput equal to a content-aware request distribution scheme even at small memory values if it has accurate and up-to-date information about the total size of the requests pending on each back-end node in the cluster. However, this approach does not seem to be practical since having up-to-date load information requires the back-end nodes to send load information continuosly to the front-end node. This would result in increased traffic on the cluster network and the front-end node may become a bottleneck. Another solution would be to send load information periodically, but again, that may not be useful as it is not very accurate when a burst of requests arrive at the cluster system. In a practical solution, the front-end uses some information, which it can update without depending on the back-end nodes. This could be the number of connections on the back-end nodes, which can be updated when a front-end selects a target back-end node for a request.

I also discovered that using a little load information in addition to the locality information can result in a huge performance improvement. When comparing the hit ratio's for each of the request distribution scheme, it was found out that for memory sizes greater than 0.2% of the total working set, even the content-blind algorithms were able to achieve hit ratios greater than 0.8. But in reality, the memory, being very expensive is only a small fraction of the total working set being hosted on a Web server. For such Web servers it is important to use memory efficiently.

Using the results obtained in comparing the request distribution algorithms, I designed a request distribution algorithm called locality aware - quality of service aware request distribution algorithm (QARD-LA), which provides differentiated QoS to clients while, at the same time, focusing on the performance improvement factors explained above. In the design, the back-end nodes are capable of providing differentiated QoS services to different classes of clients. The front-end node makes decisions about which back-end node can provide the resources requested by the client, while at the same time,

considering for locality and load balancing. The design also takes care to allocate the resources not being used by a client class to other classes. In case of overload, all the QoS requests cannot be provided with the desired demand. Those requests are then provided best effort service. I compared the designed scheme with a similar QoS scheme called load balancing - quality of service aware request distribution algorithm (QARD-LB), which provides differentiated QoS to different classes while, at the same time, trying to balance the load between the nodes in the cluster. Through simulation it was shown that, the QARD-LA was able to achieve higher throughput, better hit ratio than QARD-LB at very small values of memory sizes even if it could not balance the load as effectively as QARD-LB. I have also shown that QARD-LA was able to achieve higher revenue (or income) as compared to QARD-LB as it was able to provide more requests with the desired resource demand.

6.1 Comparison with Existing QoS Schemes for Server Clusters

Similar to Demand-driven Service Differentiation (DDSD) [ZhT01], I try to provide class based service differentiation to different classes of clients. DDSD uses a dynamic scheduler to predict the future resource demand for each class and make different partitioning decisions during system over-load and underload situations. Unlike my approach, where each node in the cluster can process requests from all existing service classes, they partition the nodes such that each partition is allowed to handle requests from one service class. This approach is very inefficient as cluster wide partitioning and repartitioning cannot be done very frequently, which makes it difficult to respond promptly to changing resource demand. In my approach the resource allocation is done at the time of selecting

a back-end node for each request. This approach can take prompt actions in case of overload or request burstiness.

Cluster reserves as proposed in [ArD00] also consider the problem of ensuring that a minimal fraction of resources be available to serve requests from a certain client community, independent of load generated by other clients. Unlike my approach, this approach does not consider back-end node selection. It is only concerned with the dynamic allocation of resources on each back-end node based on prevailing load conditions independent of the request distribution stategy employed in the cluster. This approach has a drawback similar to one of DDSD, because the resource manager responsible for allocating and deallocating resources is run periodically.

[ShT02] also consider the problem of providing service differentiation to different classes of clients. They provide a framework for class-based service differentiation in terms of resource allocation and admission control known as *Class LB*. This approach is different from mine in that they ensure a balanced distribution of requests from each class to each back-end node. At each back-end node they use a multiqueue scheduling scheme for producing high QoS yield (economical benefit). In my approach, I take document locality into consideration in addition to load information. In case of overload, this approach drops requests which will produce zero yield whereas I try to provide service to such clients based on a BES model.

6.2 Future Research Directions

Through simulation, I have demonstrated that our proposed approach can perform better than similar approaches considering only the load factor. However, several issues need to be addressed before this approach can be deployed. The first issue is related to scalabilty of the approach. Currently I do not consider that there is any overhead incurred in

making selection and prediction decisions at the front-end node. I need to make sure that the front-end does not become a bottleneck for the cluster. I also do not consider support for HTTP/1.1 persistent connections where all requests should be served by the back-end node to which the connection was handed off initially. In my simulations I set the threshold values for each class statically, but I need to use some optimization techniques to set the threshold values for each class so as to maximize the generated revenue. Furthermore, I need to see how sensitive the designed algorithm is to the CPU and disk speeds, because, being a content-aware algorithm, it should show a dramatic increase in the performance with an increase in disk speeds as compared to the load balancing algorithm which is disk bound.

Bibliography

[AlT02] Nortel Networks, "Alteon ACEdirector," *http://www.nortelnetworks.com/products /01/alteon/acedir/index.html*, 2002.

[AnP96] E. Anderson, D. Patterson, and E. Brewer , "The Magicrouter, an Application of Fast Packet Interposing," *In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, May 1996.

[ArD00] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: A mechanism for resource management in cluster-based network servers," *In Proceedings of ACM Sigmetrics 2000*, June 2000.

[ArD99] M. Aron, P. Druschel, and W. Zwaenepoel, "Efficient Support for P-HTTP in Cluster-Based Web Servers," *In Proceedings of the 1999 Annual USENIX Technical Conference, Monterey, CA*, JUNE 1999.

[ArS00] M. Aron, D. Sanders, P. Druschel and W. Zwaenepoel, "Scalable Content-aware Request Distribution in Cluster-based Network Servers," *In Proceedings of USENIX 2000 Annual Technical Conference*, June 2000.

[AvB99] Luis Aversa and Azer Bestavros, "Load Balancing a Cluster of Web Servers using Distributed Packet Rewriting," *Technical Report, Computer Science Department, Boston University*, June 1999.

[BaD99] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A new facility for Resource Management in Server Systems," *In Proceedings of the 3rd USENIX symposium on Operating Systems Design and Implementation*, Feb 1999.

[Bac99] M. Harchol-Balter, M. E. Crovella, and C. D. Murta, "On Choosing a Task Assignment Policy for a Distributed Server System," *In Proceedings of IEEE Journal of Parallel and Distributed Computing*, 1999, pp. 204–228.

[BeA97] A. Bestavros, "WWW Traffic Reduction and Load Balancing through Server-Based Caching," *In Proceedings of IEEE Concurrency, Vol.5, NO.1,pp.55-67, January-March 1997*, 1997.

[BeC98] A. Bestavros, M. Crovella and D. Martin , "Distributed Packet Rewriting and its Applications to Scalable Server Architectures," *In Proceedings of the 6th International Conference on Network Protocols*, Oct 1998, pp. 290–297.

[BrB99] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Retrofitting Quality of Service into a Time-Sharing Operating System," *In Proceedings of USENIX 1999 Annual Technical Conference, Monterey, CA*, June 1999.

[BrG98] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz, "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," *In Proceedings of USENIX 1998 Annual Technical Conference, Berkeley, CA*, June 1998.

[BrT95] T. Brisco, "DNS Support for Load Balancing," *RFC 1794*, April 1995.

[CaC01] Emiliano Casalicchio and Michele Colajanni, "A Client-Aware Dispatching Algorithm for Web Clusters Providing Multiple Services," *WWW10, Hong Kong*, May 2001.

[ChD01] L. Cherkasova, M. DeSouza, and S. Ponnekanti , "Performance Analysis of Content-Aware Load Balancing Strategy FLEX: Two Case Studies," *In Proceedings of the Thirty-Fourth Hawaii International Conference on System Sciences (HICSS-34)*, Jan 2001.

[ChG02] A. Chandra, W. Gong, and P. Shenoy, "An Online Optimization-based Technique For Dynamic Resource Allocation in GPS Servers," *Technical Report TR02-30, Department of Computer Science, University of Massachusetts at Amherst*, 2002.

[ChK01] Ludmila Cherkasova and Magnus Karlsson, "Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy WARD," *Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA*, 2001.

[ChL00] L. Cherkosova , "FLEX: Load Balancing and Management Strategy for Scalable Web Hosting Service," *In Proceedings of the 5th International Symposium on Computers and Communications, Antibes, France*, July 2000.

[CoR99] A. Cohen, S. Rangarajan, and H. Slye, "On the Performance of TCP Splicing for URL-Aware Redirection," *In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO*, OCT 1999.

[DaC97] O. P. Damani, P. E. Chung, Y. Huang, C. Kitala, and Y. Wang, "ONE-IP: Techniques for Hosting a Service on a Cluster of Machines," *In Proceedings of the 6th International WWW Conference, Santa Clara, California*, Arp 1997.

[DaK96] Daniel M. Dias, William Kish, Rajat Mukherjee, and Renu Tiwari , "A Scalable and Highly Available Web Server," *In Proceedings of the 41st IEEE International Computer Conference, COMPCON*, Sep 1996.

[DaM00] M. Dahlin, "Interperting Stale Load Information," *In Proceedings of IEEE Transactions on Parallel and Distributed Systems*, 2000.

[DiM01] J. Diamond and M. Maheswaran, "Optimization Models for Document Placement and Load Balancing on Web Server Clusters," *Unpublished Manuscript*, 2001.

[EgF94] K. Egevang and P. Francis, "The IP Network Address Translator," *RFC 1631*, May 1994.

[FoG97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer and Paul Gauthier, "Cluster-Based Scalable Network Services," *In Proceedings of Symposium on Operating Systems Principles*, Oct 1997, pp. 78–91.

[FoU02] Foundry Networks, "Foundry ServerIron Switches," *http://www.foundrynet.com/products/webswitches/serveriron/index.html*, 2002.

[GuB99] Deepak Gupta and Pradip Bepari, "Load Sharing in Distribued Systems," *In Proceedings of the National Workshop on Distributed Computing*, JAN 1999.

[HuG97] D. H. Hunt, G. S. Goldszmidt, R. P. King, and Rajat Mukherjee , "Network Dispatcher: A Connection Router for Scalable Internet Services," *IBM Research Center*, 1997.

[HuN97] G. Hunt, E. Nahum, and J. Tracey, "Enabling Content-Based Load Distribution for Scalable Services," *Techical Report, IBM T. J. Watson Research Center*, May 1997.

[JoL95] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera, "Modular Real-Time Resource Management in the Rialto operating system," *In Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[LoC96] CISCO , "LocalDirector," *http://www.cisco.com*, 1996.

[MaB98] D. A. Maltz and P. Bhagwat, "TCP Splicing for Application Layer Proxy Performance," *IBM Research Report*, 1998.

[MiM97] M. Mitzenmacher, "How Useful Is Old Information," *In Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, 1997.

[PaA98] Vivek S. Pai, Mohit Aron, Gaurav Banga, Micheal Svendsen, Peter Druschel, Willy Zwaenepoel, Erich Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers ," *In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA*, Oct 1998.

[PaB98] R. Pandey, J. F. Barnes, and R. Olsson, "Supporting Quality of Service in HTTP Servers," *In Proceedings of the Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, June 1998, pp. 247–256.

[ReS01] Resonate , "TCP Connection Hop," *http://www.resonate.com/solutions/literature/wp_cd_tcp_connect_hop.php*, Apr 2001.

[ReS96] Resonateinc , "Resonate, A Case for Intelligent Distributed Server Management," *http://www.resonateinc.com*, Dec 1996.

[ShK92] Niranjan G. Shivartri, Phillip Krueger and Mukesh Singhal, "Load Distributing for Locally Distributed Systems," *IEEE Computer*, DEC 1992, pp. 33–44.

[ShT02] K. Shen, H. Tang, and T. Yang, "A Flexible QoS Framework for Cluster-based Network Services," *Unpublished Manuscript*, 2002.

[SiM95] W. Simpson, "IP in IP Tunneling," *RFC 1853*, OCT 1995.

[VaC01] S. Vaidya and K. J. Christensen, "A Single System Image Cluster using Duplicated MAC and IP Addresses," *In Proceedings of the 26th IEEE Conference on Local Computer Networks*, Nov 2001, pp. 206–214.

[ZhB99] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer , "HACC: An Architecture for Cluster-Based Web Servers," *In Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999, pp. 155–164.

[ZhT01] H. Zhu, H. Tang, and T. Yang, "Demand-driven Service Differentiation in Cluster-based Network Servers," *In Proceedings of IEEE INFOCOM'2001, Anchorage, AK*, April 2001.