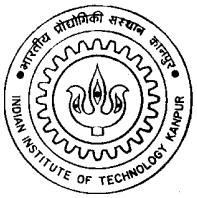
A Test bed for performance evaluation of load balancing strategies for Web Server Systems

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Puneet Agarwal



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*A Test bed for performance evaluation of load balancing strategies for Web Server Systems*", by *Puneet Agarwal*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

May, 2001

(Dr. Dheeraj Sanghi)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

(Dr. Pankaj Jalote)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Many large web sites get more than 100 million hits everyday. They need a scalable web server system that can provide better performance to all the clients that may be in different geographical regions. Higher delays and losses are common on WAN links. To provide a better service to all the clients, it is natural to have fully replicated web server clusters in different geographical regions. In such an environment, one of the most important issue is that of server selection (and load balancing). The client's request should be directed to one of the servers in a way that the response can be quick. We assume that web servers are functionally homogeneous, i.e. any one of them can serve any client request. Another important point is that this system should not require modification of any client side component or existing standard protocol.

In this thesis, we have developed a test bed to emulate the world wide web environment and compare different schemes. A large number of systems have been proposed to do this load balancing. We also propose a new scheme which is based on estimating the round trip time between the client and various server clusters. The proposed scheme is shown (through emulation) to perform significantly better than many of the existing scheme.

Acknowledgement

I would like to thank my thesis supervisors, Dr. Pankaj Jalote and Dr. Dheeraj Sanghi for their constant encouragement and innovative ideas. I am very thankful to them for allowing me to work freely in area of my interest, patiently listening to all problems, providing me every possible help instantly despite their very busy schedule. Without their support and guidance at every stage of thesis, completing this work would not have been possible for me. It has been a very enlightening and enjoyable experience to work under them.

I would also like to express my thanks towards the faculty members of Computer Science & Engineering department for imparting me invaluable knowledge and technical skills. I would also like to thank technical staff of department for providing such a nice working environment in the lab.

I also thank all my friends who made my stay here a memorable one. My all friends specially Saibal, Kingshuk, Sriram, Rajrup, Saugata, Ashish Saxena, Parthajit and Jyotirmoy were always encouraging and helpful to me.

I have used or modified many third party softwares and I would like to thank persons involved in developing software and giving suggestions in case of problems. I would like thank Andreas Gustaffson for helping me in modification of BIND, Ilia Baldine for divert sockets, Mark E. Carson for Nistnet software and Mindcraft Inc for Webstone.

Above all, I am grateful to my parents for reaching at this stage in life, it were their blessings which always gave me courage to face all challenges and made my path easier.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Number of users accessing the Internet is increasing quite rapidly and it is common to have more than 100 million hits a day for popular web sites. For example, netscape.com website receives more than 120 million hits a day. The number of users is expected to continue increasing at a fast rate and hence any website that is popular, faces the challenge of serving very large number of clients with good performance. Full mirroring of web servers or replication of web sites is one way to deal with increasing number of requests. Many techniques exist for selection of *nearest* web server from the client's point of view. Ideally, selection of best server should be done transparently without the intervention of the user.

Many of the existing schemes do only *load-balancing*. These schemes assume that the replicated site has all the web servers in one cluster. This is alright for medium sized sites, but beyond a certain amount of traffic, the connectivity to this one cluster becomes a bottleneck. So large web sites have multiple clusters, and it is best to have these clusters geographically distributed. This changes the problem to first select the *nearest* cluster and then do load balancing within the servers of that cluster. Of course, if all servers in a cluster are heavily loaded then another cluster should have been chosen. So the problem is more complex in such an environment.

Designing such system involves making decisions about how best server is selected

for a request such that user receives response of request in minimum time and how this request is directed to that server. In most strategies, a server is selected without taking into account any system state information, e.g. random, round robin etc. Some policies use weighted capacity algorithms to direct more percentage of requests to more capable servers. But few strategies select a server based on the server state and very few strategies take client state information into account. There is always a tradeoff between the overhead due to collection of system state information and performance gain by use of available state information. If too much state information (of server or clients) is collected, it may result in high overheads for collection of information and performance gain may not be comparable to overheads. So we must carefully collect only that state information that might improve performance of system as seen by clients but do not result in very high overheads.

In this thesis, we have proposed a new scheme based on collecting information about the load on each server as well as estimating round-trip time between clusters and those clients which make large number of requests.

To study the tradeoffs and impact of different parameters on a web server system, a framework is required. The framework should enable evaluation and comparison of performance of distributed web server systems. The framework should allow easy implementation of any scheme and analyze the performance of web server system with new policies.

In this thesis, we have designed and implemented a test bed to provide such a framework. We have also measured performance of few policies implemented in this test bed through emulation of world wide web scenario.

1.2 Steps in HTTP request service

Before we discuss further, it is important to understand how a HTTP request is serviced, so it is briefly discussed here. A client's request for desired object is fulfilled in following steps:

- **Domain name to IP address mapping :** The domain name present in URL must first be translated to an IP address. The client software requests its local

resolver for it, if this mapping is not in its cache. The resolver in turn returns the IP address for that domain name, that it may get from Intermediate name servers (which may have cached this mapping) or from directly from authorized DNS for that domain name either recursively or iteratively. More details about DNS mechanism can be found in RFC 1034 [26] and RFC 1035 [27].

- **Request for object to server with that IP address:** Then client software sends request for object to server having that IP address. The server may return requested object directly or it may redirect it to other server using HTTP header options or fetch the object from other server and deliver to client or may transparently forward the request to other server which replies directly to client with address of forwarding server, etc.

Thus HTTP request service path allows us to distribute requests at two levels, first at DNS at the time of resolution of domain name to server IP address, and the other at server when request reaches at that server. Any system consisting of multiple servers and some request distribution mechanism is termed Distributed Web Server System (DWSS).

Time taken for service of any HTTP request submitted by client depends on two major factors namely network conditions and server load. Even if there is a capable server system present, but the connectivity of client in terms of delay, available bandwidth or packet loss is not good, it will sees large delays. If server system is saturated with requests, time taken for service is very large. So for keeping response time minimum, web server system should take into account both the factors.

1.3 Outline

In chapter 2, we first present a brief survey of existing approaches for request distribution mechanisms. In chapter 3, design goals for system, system model taken, approach used and algorithms for each server side component of proposed architecture are discussed.

To evaluate the performance of proposed architecture and compare it with other existing proposals, a flexible test bed was designed to emulate real Internet like

scenario in which various architectures for Distributed Web Server System can be emulated with minimal efforts. In chapter 4, design goals, overview and different components of this test bed are described. In chapter 5, different algorithms implemented on the test bed and measured performance are briefly discussed and finally the performance results obtained for various schemes are compared. In chapter 6, we finally present conclusion and future extensions. In appendix, we give short description of softwares used by us.

Chapter 2

Related Work

2.1 Relation with load balancing in distributed systems

Load balancing in distributed systems has been the subject of research for last few decades. The traditional load balancing problem deals with load unit migration from one processing element to another when load is light on some processing elements and heavy on some other processing elements. It involves migration *decision*, i.e. which load unit(s) should be migrated and then *migration* of load unit to other nodes.

Both of these parts can be carried out either locally or globally. Load balancing can be classified according to the decision base and migration space [29]. If migration decision is carried out according to local load situation and that of neighbors, it is called local decision base. If this decision is based on load condition of subset of the whole network, then it is called global decision base. Similarly if load unit is migrated to direct neighbors, then it is called local migration space, otherwise it is called global migration space. So according to decision base and migration space, four different categories of schemes emerge:

- Local Decision base Local Migration Space (LDLM)
- Local Decision base Global Migration Space (LDGM)

- Global Decision base Local Migration Space (GDLM)
- Global Decision base Global Migration Space (GDGM)

A taxonomy for load balancing in distributed systems is presented in [10].

However, these approaches for load balancing are not suitable for load balancing in the web context for several reasons. First, in the web context there are multiple points for load balancing (e.g. at the DNS or at the server) while traditional techniques assume a single point. Secondly, the cost factors are not homogeneous in web and can vary a lot, while in traditional systems most servers are assumed to be generally of similar capacity and capability. Thirdly, the jobs were assumed to be compute intensive and hence the focus was to distribute the compute load. In the web, on the other hand, the load is mostly I/O oriented where caching plays a very significant role in performance and will impact the schemes. Even cost of migration of load unit and granularity of load varies for different points of load balancing. Due to these, and other reasons, it is best to consider the load balancing problem in the Web as a new problem, which requires different approaches.

In web context, which server to select has been mostly studied from client point of view, i.e. either client side DNS or client proxy or clients themselves decide which server to choose. Usually, these entities send probes to multiple servers and select best server based on probe results or they take into account previous history of responses sent by server. But these probes are usually not sufficient to accurately measure server load conditions, since load on servers can change easily with time and usually these probes can not find current load on the servers and until all clients use such softwares and there is co-operation with server side entities (it is however very difficult to reach at common method acceptable to all), they will either incur too much overhead or will not give much better performance.

Example of client themselves selecting server is Netscape [15] or *Java Applet* running at client to probe servers is [31]. In scheme proposed by Beck and Moore [7] in their I2-DSI system, DNS resolver at client side sends probes to server to select server with minimum response time. In scheme proposed by Baentsh et al [6] servers send information about other servers in hierarchy through extra http headers to client side proxy and then client side proxy selects server.

There are various proximity metrics considered for selection for best server by clients. Crovella et al [14] compare random server selection, hop count and round trip time based selection and find that RTT has relatively higher correlation with latency perceived by client. Sayal et al [25] also include HTTP latency (time measured by sending HTTP HEAD request) and all server polling in their study and find HTTP latency has highest correlation with actual server response time for other requests and present refresh based algorithms for best server selection at client side.

Client side approaches are not general, since they assume modification in client side components, some approaches even modify protocols. Thus these types of approaches can not improve performance for all the clients.

Gwertzman et al [21] first pointed out the need of creating cache server on other side of USA when demand from that side increases. Guyton et al [20] focus on hop count based metric and cost of collection of information for server selection.

Server selection at server side DNS is done based on geographical proximity approximated using client IP address or hop count information obtained from routers in Cisco's Distributed director [11]. Given that clients are distributed geographically far apart, static and relatively less costlier metrics like hop count for proximity information are not found good in study by [14]. Ammar et al [32],[17] propose local anycast resolver that is near a large number of clients, to which servers push their performance information and probing agent probes servers for path information. This proposal assumes use of anycasting domain name(ADN) and anycast resolver near clients, which once again lacks general applicability.

In next section we present a brief survey of mechanisms used for distribution of client requests.

2.2 Mechanisms for request distribution

Cardelini et al [9] classify web server architectures based on the entity which distributes the incoming requests among the servers in four classes of methods. Some of the methods in each category use feedback based algorithms and some use non-feedback algorithms as discussed in [1]. So we can categorize the request distribution

mechanisms based on entity that routes the request as follows:

- Client-based approach
- DNS-based approach
- Dispatcher-based approach
- Server-based approach
- Anycast

Last mechanism, *Anycasting* does not involve any explicit routing by web server system, but is automatically done as part of IPv6 protocol by internetworks [8].

These mechanisms offer transparency at various levels: manual selection offers *no transparency* because URLs are different; Client and DNS-based mechanisms *may* offer *URL level transparency*, i.e. URL is same but resolved IP addresses may be different; Dispatcher based approaches usually offer *IP address level transparency*, i.e. even resolved IP address is also same.

Some mechanisms are geographically scalable, i.e. cluster of servers can be either in LAN or WAN. Some approaches are fault tolerant and highly available but others are not. Some approaches require replication of whole web-site, while others allow partial replication.

2.2.1 Client-based approach

In this approach, client side entity is responsible for selecting the server so no server side processing is required for selection of server. The routing to replica is done by client software (browser) or by client-side DNS or proxy servers. So these schemes can be categorized as follows:

- **Web clients :** In this approach clients are aware of existence of replicas of same resource on multiple servers and they choose the replica themselves. Following are two schemes that utilize client software for server selection.

1. **Netscape's Approach :** This approach is taken by Netscape Navigator browsers [15]. On access to Netscape home page, browser generates a random number X between 1 and 32 and accesses http://homeX.netscape.com. Each server can have multiple homeX aliases pointing to it so that client software need not to be modified in case more servers are deployed, just changing aliases will suffice.

   This approach is not generally applicable as not all companies can control client software, it requires re-installation or change of web clients if number of aliases increase. Also, it does not guarantee server availability and load balancing of servers because if any server is down or overloaded (and the aliases has not been changed), random selection will still try to access resource from that server.

2. **Smart Clients :** In scheme proposed by Yoshilakawa et al [31], a *Java Applet* is run on the client side, whenever user accesses the Distributed Web Server System. This Applet knows all the IP addresses of servers in the System. Applet sends messages to probe node load, response time and network delays, etc., and selects the best node.

   This approach does not require client software modification and provides scalability and availability, but downloading the Java Applet requires a TCP connection, and extra probe messages cause delay and increased network traffic. Also all clients might not be capable of running the Java Applet.

- **Client's DNS resolver :** This scheme is used by Beck and Moore [7] in I2-DSI system. In this scheme, client's local DNS resolver issues probes to servers instead of web client and may choose the server based on response time or previous access performance reports from client.

  This scheme requires customized DNS and clients must also be modified for giving reports. If the server address is cached, then all requests in future will go to the same server. So load balancing may not be achieved. If caching is restricted by a lower TTL value, then we are putting additional load on DNS infrastructure.

- **Client Side Proxy :** This scheme was proposed by Baentsh et al [6]. Servers form a hierarchical structure and content replicated on each server is some part of URL name space. Each parent server in hierarchy propagates information about replicas present on direct descendents in extra HTTP headers in response to request for resource. Client-side proxy learns about replicas and next time request can go to server containing replica of resource.
  This approach requires both server software and proxy modification to give information about replica and process extra HTTP headers respectively.

All these approaches require change in client side components, which are not controlled by the e-Commerce company or the hosting ISP, So these approaches suffer from the problem of limited applicability.

2.2.2 DNS-based approach

In this approach, server side authorized DNS maps domain name to IP address of one of the nodes of the cluster, based on various scheduling policies. Selection of replica occurs at server side DNS so it does not suffer from applicability problem of client-side mechanisms. But DNS has limited control over requests reaching at server because of caching of IP address mapping at several levels viz., by client softwares, local DNS resolvers, intermediate name servers, etc. Besides the mapping, a validity period for this URL to IP address mapping, known as Time-To-Live (TTL) is also supplied. After expiration of TTL period this mapping request is again forwarded to authorized DNS. Setting this value to very small or zero does not work because of existence of non cooperative intermediate name servers and client level caching. Also, it increases network traffic and DNS itself can become bottleneck.
Several DNS based approaches are discussed in [9]and [12]. DNS based algorithms can be classified on the basis of the scheduling algorithms used for server selection and TTL values.

- **Constant TTL algorithms :** These are classified on the basis of the system state information used by DNS for server selection. The system state information can include both client and server state information, like load, location etc.

    1. **System stateless algorithms :** Most simple and first used algorithm of this type is round robin (DNS-RR). It was used by NCSA (National Center for Supercomputing Applications) [24] to handle large traffic volume using multiple servers. In this approach, primary DNS returns IP addresses of servers in the round robin fashion.

       It suffers from uneven load distribution and server overloading, since large number of client from same domain (using same proxy/gateway) are assigned same server. Also, whole document tree must be replicated on every server or network file system should be used.

    2. **Server state based algorithms :** A simple feedback mechanism from servers about their loads is very effective in avoiding server overloading and not giving IP address of unavailable servers. The scheduling policy might be to select the least loaded server any time.

       This approach solves overloading problem to some extent yet control over requests is not good because of caching of IP addresses. Some implementations try to solve this problem by reducing TTL value to zero but it is not generally applicable and puts more load on DNS.

    3. **Client state based algorithms :** In this approach, two types of information about clients, the typical load arriving to system from each connected domain (from same proxy/gateway) and the geographical proximity can be used by DNS for scheduling.

       Requests arriving from domains having higher request rate per TTL value can be assigned to more capable server. Proximity information can be used to select nearest server to minimize network traffic.

       One mode of Cisco Distributed Director [11] takes client location (approximated from client's IP address) and client-server link latency into

account to select the server by acting as primary DNS.
This approach also suffers form same problem experienced by Server state based algorithms.

4. **Server and Client state based algorithms :** Cisco Distributed Director takes server availability information along with client proximity information into account while making server selection decision. These algorithms can also use various other state estimates for server selection. Such algorithms give the best results.

- **Dynamic TTL algorithms :** These algorithms also change TTL values while mapping host name to address. These are of two types [12]:

  1. **Variable TTL algorithms :** As server load increases these algorithms try to increase DNS control over request distribution by decreasing TTL values.

  2. **Adaptive TTL algorithms :** These algorithms take into account the domain request rate (number of requests from a domain in TTL time period) and server capacities, for assigning TTL values. So a large TTL value can be assigned for a more capable server and less TTL value for those mappings that have high domain request rate.
  These are most robust and effective in load balancing even in presence of skewed loads and non-cooperative name servers, but these don't take geographical information into account.

DNS based approaches are more suitable for static replication schemes and are less suitable for dynamic replication schemes because changing place of replicated object may require change in mapping. In general these approaches suffer from limited control over request problem due to caching of resolved IP addresses at various levels.

2.2.3 Dispatcher-based approach

This approach gives full control over client requests to server side entity. In this approach, the DNS returns the address of a **dispatcher** that routes all the client request to other servers in the cluster. Thus it acts as a centralized scheduler at the server side that controls all the client request distribution. It presents single IP address to outside world, hence is much more transparent. These mechanisms can be categorized as follows:

- **Packet single-rewriting by the dispatcher :** In this approach, all packets first reach dispatcher because IP address of dispatcher is provided by DNS. All the servers in cluster have different private addresses visible within the cluster. The dispatcher selects server in the cluster using simple algorithms like round robin etc. and changes the incoming packet's destination address with the private address of selected servers in the cluster. It also maintains a list of source IP addresses for active connections and sends the received packets from each TCP connection to the same server node. Further, nodes in the cluster need to replace source address in response packets with the IP address of dispatcher.

  Although this solution maintains user transparency, it requires changes in the kernel of all the servers since packet rewriting occurs at TCP/IP level. This system combined with DNS-based solution for dispatcher, i.e primary DNS resolving host name to IP address of one of dispatcher for each cluster, can scale from LAN to WAN.

- **Packet double-rewriting by the dispatcher :** This approach is similar to the above scheme, except that all address changes are done by the centralized dispatcher, not by nodes in cluster. The dispatcher first changes each incoming IP packet's destination address to that of selected server and sends it to the selected server node in the cluster. It also needs to modify the packets on the way back to the client, i.e., now in response IP packet, it replaces the source IP address of selected server with its address. The algorithm for server selection can be round robin, random, etc.

Cisco local director selects the server with least active connections. Magic router [4] uses a application level process that intercepts all packets between client and server and modifies address and checksum fields.
This approach has advantage that it does not require modification of all nodes in cluster.

- **Packet forwarding by the dispatcher :** This approach is described in [18]. In this approach instead of IP packet rewriting dispatcher forwards packets to nodes in cluster using MAC address.

  IBM Network Dispatcher's LAN solution assumes that server nodes are on the same LAN and share the same IP address but nodes have disabled ARP mechanism, so all packets reach to dispatcher. The dispatcher then forwards these packets to selected servers using their MAC addresses on the LAN without modifying its IP header. The scheduling policy can be based on server load and availability.
  This mechanism is transparent to both client and server. No packet rewriting is required by dispatcher or servers as they share same IP address.

  IBM Network Dispatcher's WAN solution is based on dispatcher at two levels. Centralized first level dispatcher uses single-rewriting mechanism to forward the packets to one of the second level dispatchers (on WAN) for each cluster, i.e. it replaces its IP address from packets to that of selected dispatcher(each cluster has its dispatcher). Second level dispatcher (at each cluster) changes its IP address in packet back to that of first level dispatcher and forwards it to selected server on LAN using MAC addresses. Selected node responds with IP address of primary dispatcher as in the previous approach.

- **ONE-IP address :** This approach is described in [16], multiple machines in the web server system have same secondary IP address. This secondary IP address is then publicized by DNS. It is of two types:

  1. **routing-based dispatching :** In this approach all packets with ONE-IP address are directed to IP address dispatcher by the subnetwork router.

The dispatcher selects the server by applying hash function on the client IP address and then reroutes the packets to selected server using its primary IP address. Since hashing function is applied on client IP address, all packets from same client reach to same server.

2. **broadcast-based dispatching :** In this approach subnetwork router broadcasts the packets having destination ONE-IP address to all servers in web server cluster, the servers themselves compute hash function on client IP address to decide whether they are actual destination or not. It causes more server overhead.

Using simple hash function guarantees that same server will be selected for a given IP address but at the same time it is also the weakest factor in dynamic selection of server for load balancing. By changing hash function fault-tolerance can be achieved. Still hash function on client IP address is static assignment of server to each client.

- **HTTP redirection by Dispatcher**

  In this approach centralized dispatcher redirects the HTTP requests among the web server nodes by specifying appropriate status code in response and indicating the selected web server node address in its header. Dispatching can be based on load on servers or location.

  This approach is transparent to user as most browsers support it, but user can perceive little bit more delay. No packet rewriting is required in this approach but state information of the server, i.e. load, number of connections etc. should be communicated to dispatcher in this case.

  The Distributed Director [11] in second mode uses estimate of client server proximity and node availability to select the server and redirects the client to selected server. Its main disadvantage is duplication of TCP connections and hence increased delay in response.

2.2.4 Server-based approach

This approach allows two-level dispatching, first by cluster DNS and later each server may reassign a received request to one of the other server in the cluster. This solves the problem of non-uniform load distribution of client request and limited control of DNS.

- **HTTP redirection by Server**
  The approach is used in SWEB [3]. First request reaches to host in cluster using normal DNS resolution but it can further redirect request to other server. It does second level dispatching through the redirection mechanism of the HTTP protocol. This redirection may depend on the load of server or may be done in a round robin fashion. The servers need to exchange status information periodically for taking redirection decisions but this cost is negligible with respect to traffic generated by client requests. Its main disadvantage is duplication of TCP connections and hence increased delay in response.

- **Packet Forwarding by Server**
  In this approach, first level scheduling is done using round robin DNS mechanism, the second level dispatching is done by packet rewriting mechanism that is transparent to users. So first request reaches to any node in cluster, if that node figures out that other node is better for serving this request, node uses MAC address to reroute the packet to selected sever.

  It does not require HTTP request redirection hence it is better in terms of latency time. The server selection can be stateless i.e. based on hash function or based on load information on servers. If loading information is used for rerouting, server need to exchange load information among themselves. Also this scheme can work with both LAN and WAN based solution.

- **Akamai's Approach**
  Akamai's approach [2] is very different. In their approach, URLs of objects embedded in HTML page, like images, Java Applets, multimedia components etc., are modified by proprietary software *Launcher* running at server, to the

URLs of the objects available at any Akamai server nearest to client. It is claimed that these embedded objects comprise nearly 70% of typical page in overall bytes. A map of current Internet traffic conditions, the loads of all Akamai servers worldwide, and the locations of Internet users is built for selection of server. This map is updated once per second. While making selection of server, it is made sure that no server is overloaded and number of servers containing replica is proportional to number of requests for the object. This approach is very useful when page contains large multimedia objects. It requires protocol for getting information about other servers distributed geographically, and client location. It scales geographically well but it also requires pages to be modified according to the client location.

2.2.5 Anycast

In IPv6, an *anycast* service [8] will be supported. This service assumes that the same IP address is assigned to a set of hosts, and IP router has path to its closest host in routing table. Thus different IP routers have paths to different hosts with the same IP address.
This approach automatically selects the closest host, thus load distribution causes no overhead. But it also implies almost static replication since changes in routing table take time. Which can be solved in future through Active Networks, in which simple program injected by application can be executed at routers.

These mechanism have their relative pros and cons. Client side approach does not require any server side processing but suffers from limited applicability problem. DNS based approaches suffers from problem of limited control over client request due to caching and non-cooperative name servers. They provide coarse level control over client request but these approaches do not suffer from single point of failure problem which is present in Dispatcher based approaches. Dispatcher based approaches give finer level control over client request. Packet forwarding approaches are most suitable for LAN based solutions and can scale to WAN solution. Server based approaches offer fine grain control and do not suffer from single point of failure

problem but redirection causes increase in latency period.

Our focus is on a general scheme that can be fully implemented at server side and can be very easily deployed with currently used infrastructure and standard protocols. Hence we do not consider client side approaches and do not assume existence of any support or special component or modified protocol running at client side. We consider whole server architecture for collection of metrics required for selection of server, role of each entity and method of request distribution.

Chapter 3

Proposed Architecture for Web Server System

In this chapter we discuss design goals for system architecture, system model used and algorithms at each server side entity.

3.1 Design goals

A Distributed Web Server System (DWSS) consists of a large number of servers with some mechanism to distribute the incoming client requests among those servers. We have the following design goals for the DWSS architecture:

- Components used should be *compatible* with current protocol and network elements, i.e. they can be deployed in current infrastructure and protocol suite very easily.
- It should not require change of components at client side or components on which website administrator has no control, i.e. *change in only server side components* is allowed.
- System should be *geographically scalable*, i.e. more servers in clusters can be added when needed in LAN environment and besides that more clusters (that may be geographically far apart) can be added in web server system on WAN.

- System should give *better performance in terms of latency perceived at client side*, i.e. time lag between request submission by user and content reaching at client side software should be minimized.
- System should be *user transparent*, i.e. single virtual interface to access website should be provided at the URL level, request should be directed to appropriate server automatically by web server system.
- System should be *fault tolerant*, i.e. system should continue working (may be with degraded performance) even if some servers or clusters fail or taken off-line.
- System should *avoid overloading of any server*, i.e. requests beyond capacity of any server should not reach to it, since it may result in crashing of servers.
- System *should not incur too much additional overhead* for its functioning, in terms of computation required or network traffic generated.

3.2 System model

Our system model taken by us is shown in Figure 3.1. Different steps in HTTP request service are shown in this figure. Client software first asks its local resolver for IP address of web server, if local resolver or intermediate resolvers do not have this mapping or TTL has expired, this request reaches to server side authorized DNS in step 1 and DNS replies with IP address of front node of one of several cluster (selected according to algorithm, which we discuss later) in step 1.1. In step 2, client software or some entity on behalf of client (client proxy or gateway) sends request to front node of that cluster using obtained IP address in step 1. Front node decides which server in the cluster should serve the request (algorithm for selection is described later) and request is forwarded to that server by front node in step 3. Finally, in step 4, selected server replies with request object on behalf of front node.

We have chosen cluster based model because it creates additional level for system state information collection and gives full control over dispatching of each HTTP

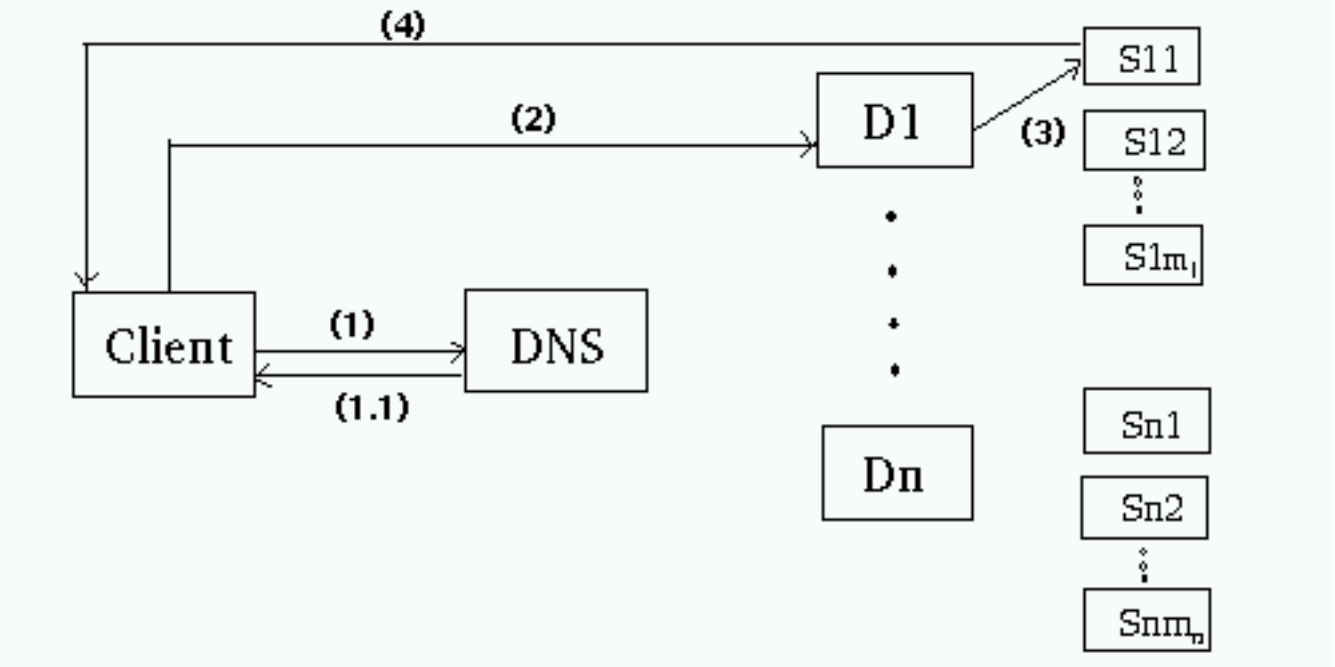


Figure 3.1: System Model

connection. Besides, our assumption is that clients are geographically distributed in distant parts of world and company can place each cluster at strategic location near its customers, where they can serve customers better. This model also allows website administrator to change number of servers in any cluster as well as change the number and location of clusters easily.

This model allows us to collect finer level information about each server at the cluster level and aggregated information about each cluster can be passed to entity(DNS in our case) requiring this state information for making request distribution decisions at coarser level.

3.3 Request distribution strategy

Our aim is to assign each client request to the *best* server such that client experiences minimum latency between HTTP request and reception of requested object.

First level decision can be taken by DNS itself, DNS can resolve IP address of cluster which can give better service to this client. Parameters affecting delay in service of HTTP request are load at selected server (and hence cluster) and path characteristics between client and server. So to take this decision, DNS should have recent cluster state information and proximity of client with clusters. DNS has information about client IP address and cluster IP addresses. Since clusters are under control of website administrator, they can provide any state information required by DNS. Since only server side components can be modified, they will have to gather the proximity information themselves. There are various metrics to measure proximity between the client and clusters. Some metrics are:

- Geographical distance between cluster and client
- Network distance in hops between cluster and client
- Round trip time (RTT) between cluster and client
- Available bandwidth on path between cluster and client
- Response time of any prior web document fetch
- Latency of any prior web document fetch

Geographical distance is significant only when time taken for transmission of requested object and propagation delay on wire are comparable, i.e. propagation delay is also significant. Propagation delay is significant for very large distances even at speed of light (can be 100s of milliseconds). But transmission media used for long distances (usually optical fiber) has very low delays and if satellite communication is used for even local connections, geographical distances may not correspond to actual delays on network. Nevertheless, it results in lesser traffic on long distance lines and usually corresponds to lower delays in practice. Geographical distance can be *approximated* by IP address of client if such database is available. According to RFC 1466 [19] different IP address ranges were allocated to different geographical regions to keep routing tables shorter. Using higher 8 bits of IP address only, geographical region of client can be approximated.

Network distance in hops is also a good metric and can be obtained from routers. But it does not take into account bandwidth available in path, current traffic on the path between cluster and server. In short, available bandwidth for transfer on path and delays in each hop are not taken into account. It is usually a static measure of proximity, since as found by Paxon [28] that 68% routes on the Internet are stable for at least a week and 87% routes on Internet are stable for at least six hours. Also, studies by Crovella et al [14] have found that the hop count has very low correlation (0.16) with response time (measured at client side). So it does not seem very good metric to use.

Round trip time is another metric that can give better and relatively accurate delay experienced in path and to some extent, a lower RTT indicates higher available bandwidth. However, it is very dynamic in nature, it changes quickly over relatively short period of time. It has much more variation for different clusters compared to hop count, it gives better path information between client and cluster. On the downside, it is relatively costlier to measure and requires more frequent refreshes.

Measuring bandwidth, by using tools like pathchar [22] or even using other more efficient current techniques [23], generates lots of additional network traffic and takes long time, so use of this metric is not practical.

Last two metrics can be used only after a number of clusters are tried (which result in degraded performance for client) and a huge database is maintained. Still, load on clusters can change over time and older information may not predict good cluster. These metrics are really useful for client side server selection only.

After comparing these metrics for client-cluster proximity, we conclude that RTT is the best metric to use for getting path information. It requires periodic refresh and is relatively costlier to measure (compared to hop count or geographical information) but it provides current and better network characteristics information. So we should try to limit overheads in measuring it. Crovella et al [14] found in their study that when used at client side it resulted in less than 1% additional network traffic and gave very good results when three ping messages were used to measure RTT information.

To further minimize overhead, instead of all clusters measuring RTT for each client, we propose to do the measurement only for a small subset of clients with

very high request rate. Arlitt et al [5] find out that 75% of total HTTP requests to any server come from 10% of networks. So if we collect information about only very high request rate generating clients, we can use that information for all clients on the same network. We can further limit number of clusters which should measure RTT based on geographical information (approximated by use of client IP address) and having less load to certain maximum number(say at most 3).

In our approach, each server gives state information to front node in the cluster and this aggregated state information is used for assigning requests within the cluster and is propagated to DNS in aggregated form to make coarse grain (per client IP based) request assignment to cluster. More details can be found in section 3.6. We gather this proximity information once high request rate is reported by cluster to DNS, so it does not delay reply from DNS, however first reply for even those clients is based on geographical proximity information approximated using IP addresses (it is used for all clients, who either do not generate large number of requests or query DNS first time after long interval).

3.4 Overview of architecture

Web server system consists of many clusters distributed geographically all over the world placed at strategic locations, similarly clients are also in different geographical regions. Thus it enables us to take into account variation of request rate from each geographic region.

Our approach is to dynamically distribute requests based on current system state information. All servers in cluster report state information to front node and front node uses this information to distribute individual client requests (each TCP connection) coming to cluster among servers in cluster intelligently. Front node reports aggregated cluster load information to DNS like a single node of high capacity. Which once again uses this information to resolve IP address of a cluster for queries from client to provide them better service in terms of perceived delay. Collecting only server state information is not sufficient, servers also collect number of requests coming from each IP address and send to front node which aggregates

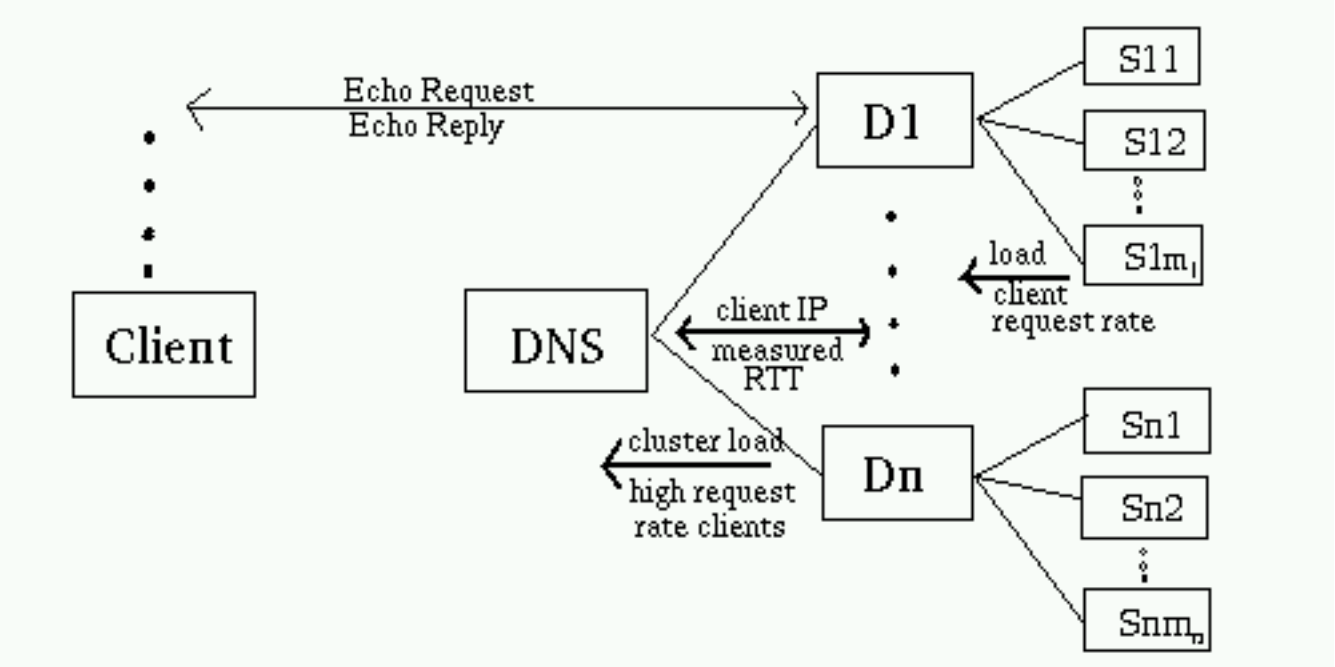


Figure 3.2: Additional messages exchanged among components in DWSS

this information and reports IP addresses of clients having very high request rate to DNS. It has been found that more than 75% requests can come from 10% of networks. Using request rate information, DNS asks few clusters to collect cluster to client proximity information only for those clients. Cluster to client proximity is found by sending ICMP Echo request messages to clients. Additional messages sent among entities are shown in Figure 3.2.

For client request distribution inside each cluster, IP packet forwarding by changing destination IP address of request packets only can be used as shown in Figure 3.3. Every IP packet that reaches at front node for HTTP connection is diverted at IP layer before delivery to TCP layer. A program running at the front node selects server for this client based on client IP address and server load information, address of selected server is filled in destination address field of IP packet and packet is re-injected back on network, so it reaches the selected server. At server this packet is once again diverted and destination address is set back to IP address of front node

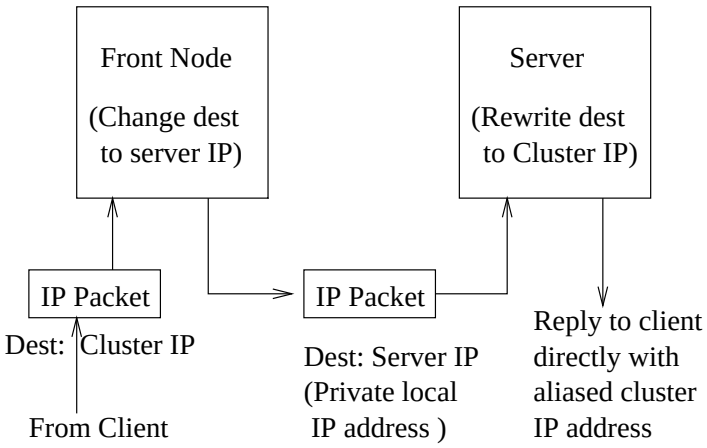


Figure 3.3: One way distributed IP packet rewriting mechanism

and is re-injected in TCP/IP stack of server node. Each server has secondary IP address (ARP disabled) same as IP address of front node, so HTTP server accepts this packet and response packets directly go to client from selected server without doing any additional modification or delay. This results in additional delay of about one millisecond for each incoming packet, if servers and front node are on same LAN. Since this packet forwarding can be done at application layer, it was chosen for emulation, however in actual system, packet forwarding inside kernel using MAC address can be done or dedicated hardware can be used for more efficient dispatching.

3.5 Algorithms

Load balancing is done at two places in path of HTTP request service, first at the DNS level and secondly at the front node of cluster. DNS tries to balance load on clusters by providing IP address of appropriate cluster's front node. When request reaches the front node, it balances load amongst the servers in that cluster. In exceptional cases when cluster is overloaded (due to uneven request rate from clients and caching of DNS entries), HTTP requests can be redirected to other

lightly loaded cluster(s).

Within each cluster, every server periodically sends its load information to front node, which sends aggregated load information about cluster to DNS. This load information transfer can take place aperiodically too if load condition changes suddenly at any component, say any server becomes overloaded.

A number of system state information parameters are collected by each server, for example, system load averages, system and user cpu utilization, free RAM, Buffer RAM, number of disk accesses, free swap, number of processes, number of requests served in last 64 seconds and number of bytes transferred in last 64 seconds. Using average number of connections sent (dispatched and currently active) to particular server in past predefined time interval and its load condition in that time interval (a user defined function depending on bottlenecks present) capacity of each server, i.e. average number of connections it can serve without significant increase in response time is dynamically estimated and updated with every load update from server by front node. Similarly every front node aggregates load information of every server and informs available free capacity of whole cluster to DNS periodically.

We describe algorithm below at each component (DNS, front node and servers in each cluster).

3.5.1 Load balancing at DNS

In response to query from client for resolving domain name, DNS returns IP address of server. All requests are sent to server having that IP address for time period called Time to live (TTL). After expiration of TTL, query is once again sent to DNS. Since within TTL period all requests from that client (or its gateway) are sent to the same server, if number of request generated by that client are higher than others it can create load skew. Aim should be to assign clients having high request rates to servers having higher capacity. Again TTL value should be small because load skew is created by these clients.

For getting request rate (number of requests in unit period), servers (or front nodes of clusters) should send this information periodically to DNS. We distinguish between clients based on their request rates. Servers send information of request

rate only when client request rate is higher than a threshold. DNS instructs few possibly nearest clusters (having remaining capacity higher than request rate) to get round trip delay to client.

For clients having high request rate, we maintain information about their request rate, list of few candidate servers(say 3) having enough remaining capacity at time of RTT probe with RTT, time stamp of last RTT probe.

Procedure recvmsgs is procedure responsible for receiving messages of different types and dispatching these messages to appropriate handler functions depending on type of message, pseudo code below shows main messages received :

```
procedure recvmsgs
{
 Input: Socket for receiving messages
 Output: None

 read message from socket and determine type of message

 switch(message type){
        case load_info:
        /* Message from front nodes about load information on each cluster */
            call update_load
            break;
        case request_rate:
        /* Message from front nodes about request rate of clients */
            call update_requestrate
            break;
        case ip_request:
        /* Message from DNS for preferred IP address of client */
            call resolve_ip
            break;
        case rtt_reply:
        /* Message from front nodes about RTTs between cluster and clients */
```

```
            call update_rtts
            break;
}
```

Procedure update_request_rate is called when front node sends this request rate information to DNS.

```
procedure update_request_rate
{
 Input: Client IP addresses, request rate
 Output: None

 for each IP address of client (or its gateway) {
    if(no request rate available for this IP)
       add request rate record for this client with current time stamp
    else
       update request record for this client with current time stamp

    if(no candidate server in list or time stamp of probe is too old)
        send_probes_for_rtt(Client IP)
 }

 update average request rate information.
}
```

If no request rate information about a client IP is received for few periods of request update then that entry is deleted.

Procedure send_probes_for_rtt adds IP address of client for sending probe for measuring rtt to list of new nearest and not overloaded clusters.

```
procedure send_probes_for_rtt
{
 Input : IP address of client to probe
```

```
 Output: None

  select few clusters nearest (approximated using IP address) to client having
  remaining capacity > request rate of client

  for each cluster in above list
          add client IP for sending request for rtt probes for this cluster

  update probe timestamp for client with current time
}
```

Actual message containing Client IP addresses is sent to each cluster periodically after every fixed interval or sufficient number of clients are already queued.

Procedure update_rtts is executed when message from cluster front node about information of round trip time between them and client is received.

```
procedure update_rtts
{
 Input: Cluster IP, Client IP, rtt, number of successful rtt probes
 Output: None

 if(number of candidate servers is less for Client IP)
      add_candidate(Client IP,Cluster IP,rtt,num probes)
 else if(any candidate server has higher rtt in candidate server list
         or had less number of successful probes)
      update_candidate(Client IP,Cluster IP, rtt, num probes)

}
```

add_candidate and update_candidate keep a list of rtt records in ascending order of round trip time and number of successful rtt probes for given client IP.

Procedure update_load is executed when message from front node of any cluster about load information is received.

```
procedure update_load
{
 Input: IP address of cluster's front node, capacity, load
 Output: None

 find record for node
 update load information of cluster
 update available free capacity of cluster and whole system
}
```

Finally clients request for host name to IP address resolution.

```
procedure resolve_ip
{
 Input: IP address of client (or its gateway, i.e. firewall etc.) and domain name
 Output: IP address of front node of cluster

  if (information about client request rate is available){
      if(probe time stamp is too old)
             send_probes_for_rtt(client IP)

      find list of clusters sorted on previously probed rtt to client
      for each cluster in list in ascending order of rtt
         if(available capacity of cluster > request rate of client){
                 reduce available capacity of cluster by client request rate
                 return(Cluster IP address);
         }
      /* If all servers probed are overloaded */
      send_probes_for_rtt(client IP)
  }
  else{
        set request_rate to average request rate of all clients.
        find list of nearest clusters sorted on nearness approximated by IP address
```

```
        for each cluster in list in ascending order of proximity
           if( available capacity of cluster > request rate of client){
               reduce available capacity of cluster by client request rate
               return(Cluster IP address);
           }
  }
  /* If no cluster is yet selected, all servers are overloaded */
  select cluster in proportion to free capacity
  return(Cluster IP address)
}
```

3.5.2 Load balancing at front node of each cluster

First front node collects information about request rates from each client IP, then periodically it sends request rate information of only those clients which have high request rate to DNS.

Similar to DNS, front node also receives different types of messages and invokes appropriate message handler based on type of message, main messages are server load information and client request rate from each servers in cluster, request for measuring RTT to client from DNS and it also selects server in cluster for each new TCP connection from client and rewrites destination address of IP packets coming from clients with selected address.

Each server periodically (at large intervals of order of minute) sends request rate information of clients in terms of number of requests by that client. On receipt of request rate update message, receive_request_rate procedure is invoked.

```
procedure receive_request_rate
{
 Input: Client IP addresses, requests
 Output: None

 for each Client IP address
    update_request_rate(Client IP,number of requests)
```

```
 update global request rate information
}
```

update_request_rate creates new record or finds record for given client IP and aggregates request rate information about each client.

Periodically cluster sends aggregated request rate information of clients which generate high number of requests than average client.

```
procedure send_request_rate
{
 Input: Client IP and their request rates
 Output: None (sends this info to DNS)

 calculate Threshold based on average request rate

 for each Client IP having request rate > Threshold{
    add Client IP and request rate in queue
    if(queue is full)
        send queued request rate information of clients to DNS
 }
 send queued request rate information of clients to DNS
}
```

Front node receives detailed load information from each server periodically. Using average number of connections sent to it in that predefined interval and obtained load information from server, front node estimates number of connections server can serve, i.e. capacity of server. This estimate is updated with every load update from server.

```
procedure receive_server_load
{
 Input: Server IP address, load
```

```
 Output: None

 find record for server using IP address and update server load
 estimate and update number of connection server can serve
 update cluster's load information and available capacity
}
```

Cluster sends aggregated load information periodically to server or when load condition changes significantly.

When DNS requests for measuring RTT between client and Cluster, following procedure is executed.

```
procedure receive_probe_for_rtt
{
 Input: Client IP addresses
 Output: None

 for each Client IP address in list
      send predefined number of echo requests to client periodically
}
```

Clients reply with Echo reply for each echo request, RTT is measured and averaged. Average RTT along with number of successful probes are sent to DNS periodically.

Finally it forwards requests to servers in cluster in proportion to remaining capacity of each server,

```
procedure forward_request
{
 Input: IP packets from clients for HTTP request
 Output: IP packets with destination address of selected server

 if(connection already exists for this client IP and port){
```

```
    if(packet is fin)
          move this connection record to a list where it will be recycled after few
          minutes

    update time stamp for this connection
    write IP address of server in destination field and re inject on network
 }
 else if(packet is syn){
       select servers in proportion to their remaining capacity
       create new connection record with current time stamp
       write IP address of server in destination field and re-inject on network
 } else
          drop this packet

 if(load on each server > capacity and least loaded cluster list not empty)
        redirect request to other clusters in proportion to their free capacity
}
```

All the connection records for connection on which there was no packet transmitted from source for a long time are also freed periodically.

3.5.3 Support at each server

Each server sends its load information to front node periodically or when its load condition changes significantly.

```
procedure send_server_load
{
 Input: Current load
 Output: Sends load information to front node

 get current load information from system
```

```
 send_load_to_front_node(load)
}
```

Each server also sends client request rates to front node periodically however at longer interval (order of minute).

```
procedure send_request_rate
{
 Input: Client IP and their request rates
 Output: None (sends this info to front node)

 read html access log file and aggregate number of requests from each client
 send_request_rate_to_frontnode(Client IP, request rate)
}
```

Also each server has secondary aliased IP address same as front node's IP address so when packet is received using other IP address, this packet should be re-injected back in protocol stack with changed destination IP address of front node.

```
procedure change_destination_address
{
 Input: Incoming IP packets for HTTP connection
 Output: IP packets with changed destination address

 for each incoming IP packet for HTTP connection
     rewrite destination address to IP address of front node(and secondary IP)
      and re-inject it back in TCP/IP stack
}
```

Thus IP packets received by front node are forwarded to server using local private IP address of server and then server rewrites dest address back to cluster IP address and to tcp layer it seems that this packet came with destination address of aliased secondary IP address directly.

Chapter 4

Test bed for Measuring Web Server System Performance

We needed a framework for studying tradeoffs and impact of different parameters on a web server system, this framework was required to test performance of Distributed Web Server System proposed by us and compare its performance with other architectures proposed earlier e.g., round robin, random, weighted etc.

To compare various policies for request distribution at server side, we designed and implemented a test bed which tries to emulate real network scenarios and follows all steps in HTTP request service. In fact, all standard components used in the Internet are used in this test bed, for example, BIND (Berkeley Internet Domain Name Server) is used for DNS and Apache web servers. We have used *Webstone* [30] for generating HTTP requests. We have modeled WAN delays and bursty packet losses which are common on Internet links. All machines used are Pentium PCs running Linux operating System.

In this chapter, we first describe design goals, and then discuss our assumptions. After a brief overview of the test bed, we describe request distribution mechanisms used at the front node and DNS. Lastly we describe various components of the test bed.

4.1 Design goals

The test bed was designed to facilitate easy measurement of various parameters of web server performance like average response time for requests and the throughput of Web Server system. While setting up the test bed following goals were kept in mind:

1. The test bed should *emulate real Internet scenario* in the lab environment. It should use standard components and follow standard protocols used in Internet.

2. Test bed should be *general* enough so that different policies for request distribution at front node and DNS can be easily incorporated in this test bed. Thus it should make comparison of different schemes very easy.

3. The test bed should be *flexible* enough to modify only selected components without needing many changes in other components.

4. Servers should pass their state information to front node and front nodes should pass cluster state information to DNS so that various *dynamic policies* based on system state information for request distribution can be implemented and compared easily.

5. Design of test bed should be such that it does not constrain or fix the number of servers, clusters and clients to be used in the test bed.

6. It should only focus on distributed web sever system implementation and we should be free to use standard benchmarking software like "Webstone" for testing the performance of system.

4.2 Assumptions

Since test bed was created for emulation of Internet environment in lab, we made the following assumptions:

1. In IP packet forwarding mechanism, it was assumed that each IP packet will contain TCP header, i.e., IP packets are not fragmented. In Linux, higher layers indeed use maximum transfer unit information so that packets do not need fragmentation and reassembly in LAN environment.

2. To avoid any central entity like router from becoming bottleneck, packet losses and delays in one direction are introduced by front node when packets reach to web servers and by clients themselves when packets arrive from servers for them.

3. We have implemented one way distributed packet rewriting for request distribution at front nodes and all servers have to rewrite incoming IP packets for HTTP connections. We assume that overhead of rewriting incoming IP packets for re-injection in TCP/IP stack with aliased secondary IP address is negligible.

4.3 Overview of test bed

Different steps for measuring performance of distributed web server system are shown in Figure 4.1. Load generator (any third party benchmark program) runs on nodes at client side and generates HTTP requests to distributed web server system. Distributed web server system is part of test bed and its components are modified corresponding to load balancing strategy used in web server system. Additional software components running at these nodes collect statistical information, which is collected and processed. After processing this statistical data, performance is analyzed and results are presented.

Web server system in test bed uses the same general hierarchical structure shown in Figure 3.1. This model allows one to emulate behavior of multiple networks in different geographical regions. Single server can be used instead of one cluster containing front node and multiple servers, so this test bed allows us to emulate cluster based as well as independent server based architectures or web server systems containing mixture of both.

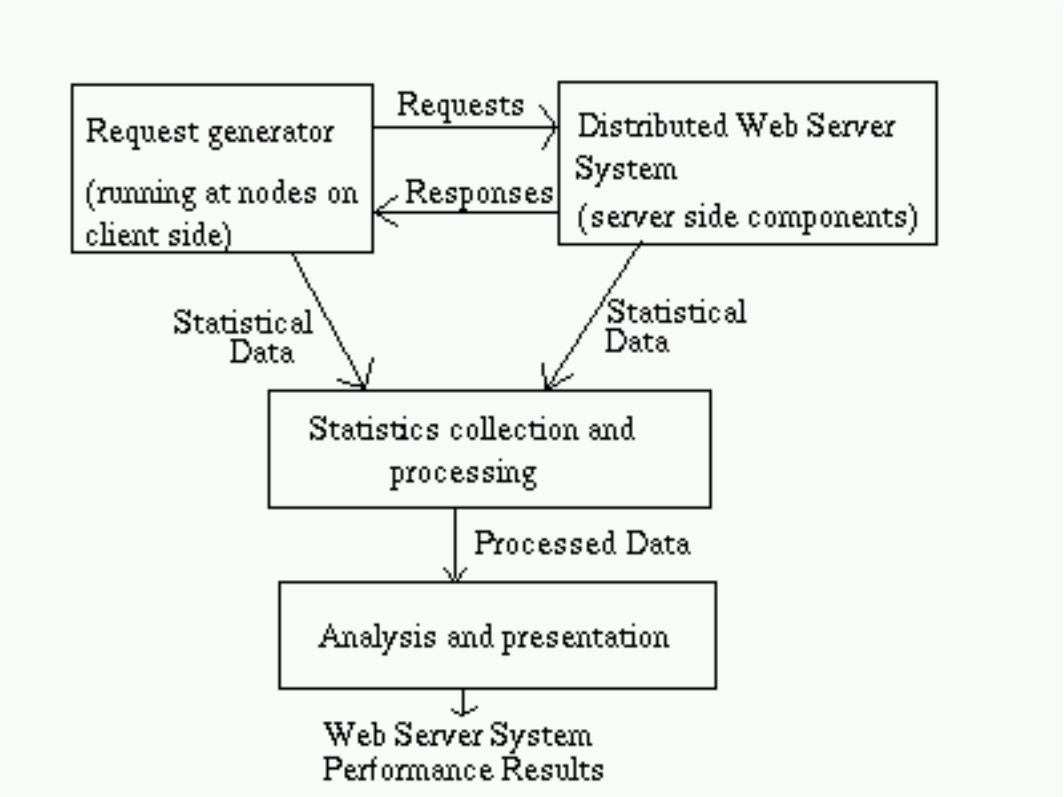


Figure 4.1: Block diagram of Test bed

Web server system consists of many clusters distributed geographically all over the world placed at strategic locations in possibly different time zones. Similarly client domains modeled by multiple client processes on one or more clients also are divided in different geographical regions. Thus it enables us to take into account variation of different parameters like delay, loss, request rate etc. from each geographical region.

Since we have tried to emulate real Internet like system and used standard components and protocols, we expect to use same test bed for measuring performance of almost every load balancing web server system with minimal modification in some components corresponding to mechanism used in the system. Below we briefly describe software components that run on different components of test bed

4.3.1 Software component at each server

Software running at each server in cluster collects system state information like load average (i.e. average number of process ready to run in last 1, 5 and 15 minutes), CPU and memory utilization, number of active connections, number of server processes running to handle client requests (with Apache server, the number of processes running to handle requests is automatically determined based on number of requests).

This software periodically obtains system load at short intervals (every 500 msec) and if load has changed since last update considerably, change in load is propagated to front node. Otherwise, if load does not change appreciably, still load update is sent every four second (8 clock ticks of 500 msec) to front node as heart beat message to inform that it is still alive and update its load information. To maintain consistent view of load information of all servers in cluster, all servers send this periodic load update at approximately the same time to front node.

Using web server access log (which is assumed to be in the standard Common-Log Format [13]), number of requests from each client domain (IP address of client) is determined and this information is propagated to front node. This information is collected and sent periodically at larger intervals (128 ticks of 500 msec, i.e. every 64 seconds).

At every server, every incoming packet for HTTP connection is diverted from TCP/IP stack and after changing destination address back to that of IP address of front node, it is re-injected back in TCP/IP stack. Now this packet is received by HTTP server, running at that server, as normal packet coming to it from interface having IP address of front node.

4.3.2 Software components at front node of each cluster

Front node is responsible for distributing requests coming to cluster, it takes into account load on each server and previous request rate of client (if available), before dispatching request to any server. We are using only incoming IP packet destination IP address rewriting to dispatch client requests among servers in cluster. So selected server depends on client IP address (hence its previous request rate) and server

load. We use IP firewalling mechanism that in turn uses Berkeley packet filter facility inside the kernel at low level to filter packets coming for HTTP port and using divert socket (that stops packet from going up in TCP/IP stack) packets are received by application program, which selects server and writes selected server's IP address in destination address field and re-injects that packet back in the network.

Front node also receives load updates (heart beat messages) and asynchronous alarm messages about overload and underload situations of server. It also receives client request rate in the last 64 seconds from each server and aggregates this information. Each front node is in sync with DNS for alarm ticks. So all front nodes in the system, receive load updates at almost the same time, aggregate and send the aggregated load information to DNS. Thus DNS receives latest and consistent information about all clusters. Front nodes also send request rate information of clients having very high request rate (above the average request rate of client domains) to DNS but this information can put more load on DNS so this information is sent in distributed manner by different front nodes every 64 seconds.

Separate optional application on front node also receives requests from DNS to send ICMP echo requests to select clients. It sends ICMP echo request messages to those clients and reports RTTs between clients and that cluster. Since only small percentage of clients are sent ping messages, and clients whose recent RTT information is available are contacted only after a refresh time interval in our proposed architecture, load on the front node due to it is not expected to be high. This responsibility can be handed over to the least loaded server in cluster easily.

4.3.3 Software components at DNS

DNS may use load information of each cluster, client request rate and proximity information to resolve IP of any cluster (i.e. IP of front node or shared secondary IP address of each server in cluster). Current implementation of domain name server (BIND-9.1) do not have any support for weighted capacity of IP resolution or any other dynamic policy based on current load, etc. It only supports random selection of IP address for Address query when multiple IP addresses are present for single server as specified in RFC 1034 [26] and RFC 1035 [27]. We have extended BIND

for this purpose.

For selection of desired IP address depending on client IP address, we have created a separate application that can run on the same DNS machine or any other machine. BIND has been modified to send client IP address to this application, which selects cluster IP address for that client as per policies implemented and BIND returns that IP address to client. That application may select IP address of cluster based on loads of server and proximity approximated by IP addresses of clusters and clients, if no real proximity information(e.g. RTT) is already available for any client, e.g. if client is sending request for the first time or after a long time when its information is deleted or the client does not generate enough requests. Optionally, this application can send queries to front nodes for different clients, it then receives and automatically updates RTT between clusters and client.

DNS receives load updates periodically from each cluster. If load on cluster is very high or load information is not received, DNS may not resolve IP address of that cluster further till load conditions return to moderate level on cluster depending on policy used. DNS also receives IP addresses of high request rate clients from each cluster at larger interval (each cluster sends this information every 64 seconds). This information can be used in different policies if desired, for example, in architecture proposed by us, DNS selects a subset of clusters (3 clusters at most) which are nearer to client (approximated using IP addresses) and are not overloaded. Thus DNS collects client IP addresses whom different selected clusters should ping to measure RTT. Requests to measure RTT to clients are sent by DNS to clusters. These clusters measure RTT to each client and return measured RTTs to DNS, DNS updates proximity information for each client in hash table and for next address resolution reply to client, this proximity information can be used.

4.4 Request distribution mechanisms

We have implemented mechanisms for request distribution at two places, at DNS and at the front nodes.

4.4.1 At DNS

At DNS, using a separate application, which runs along with modified BIND server, desired cluster IP address for different clients can be selected according to desired policy. We have already implemented four policies : random, round robin, weighted and nearest server selection (proposed by us). In our application, there is a method select_cluster which takes input client IP address and selects cluster as per policy specified. New policies can be implemented very easily by modifying this method.

All available information about clusters and clients having high request rate (if present) is accessible easily using their IP addresses. Information about all clusters can also be obtained sequentially.

4.4.2 At Front nodes

At front node, using our application each new TCP connection from clients for HTTP request can be scheduled on desired server. Similar to DNS, we have implemented three policies for server selection at front node : random, round robin, weighted round robin (based on current load of servers). By modifying a method called select_server which takes client IP address as input and returns server IP address to which this new connection should be forwarded, scheduling policies can be easily changed. Currently, distributed IP packet rewriting mechanism is used, so only client IP address and TCP port number of client side can be used to determine which server to select.

All available information about servers and client request rate information (for predefined time interval in past and average) is accessible easily using their IP addresses. Information about all servers can also be obtained sequentially.

4.5 Experimental setup

We have setup a test bed having 3 clusters on different logical networks modeling three different geographical regions. Each cluster has one front node and two servers connected to front node for that cluster. Servers are configured to have aliased

secondary IP address same as cluster IP and have local private IP address that is used for IP packet forwarding by the front node.

We have used ten clients to generate requests to web server system. Clients were also assigned IP addresses in such a way that clients in same geographical region had higher order seven bits as mentioned in RFC 1466 [19] describing guidelines for management of IP address space. Using this RFC, we modeled three geographical regions for clusters - region1 as Europe (machines had IP addresses in 194.*), region2 as North America (machines had IP addresses in 198.*) and region3 as Pacific Rim (address with 202.*). Similarly three clients each were present in region1 and region2 and two clients in region3. We also had three more clients in other regions which represent mix of client not falling in either of three regions. A DNS was also setup to resolve IP addresses of clusters. Actual test bed setup used for performing experiments is shown in Figure 4.2.

To model WAN effects, artificial delays and packet losses were introduced using Nistnet software. Half of delay (in specified range) and losses occurred in one direction and half in the reverse direction. Front nodes introduced delays and packet losses for packets transmitted by clients and clients introduced similar delays and losses after receiving packets from servers but before giving it to the higher protocol layers.

We have configured lower delays for IP packets sent and received between clients and servers in the same geographical region and relatively higher delays for packets between clients and servers in different geographical regions. These delays were generated randomly within specified range (say, 10-50 ms round trip delay in the same region and 50-250 ms delay across the regions).

Similarly we configured lower packet losses with higher correlation between drop of packets to model bursty lower packet losses in small distance links for links in same geographical region and higher packet losses with high correlation between successive packet drops for links across different geographical regions (e.g. 5% loss with .9 correlation on links in same region and 10% loss with .85 correlation on links connecting different regions).

More details about experiment are discussed in the next chapter containing results.

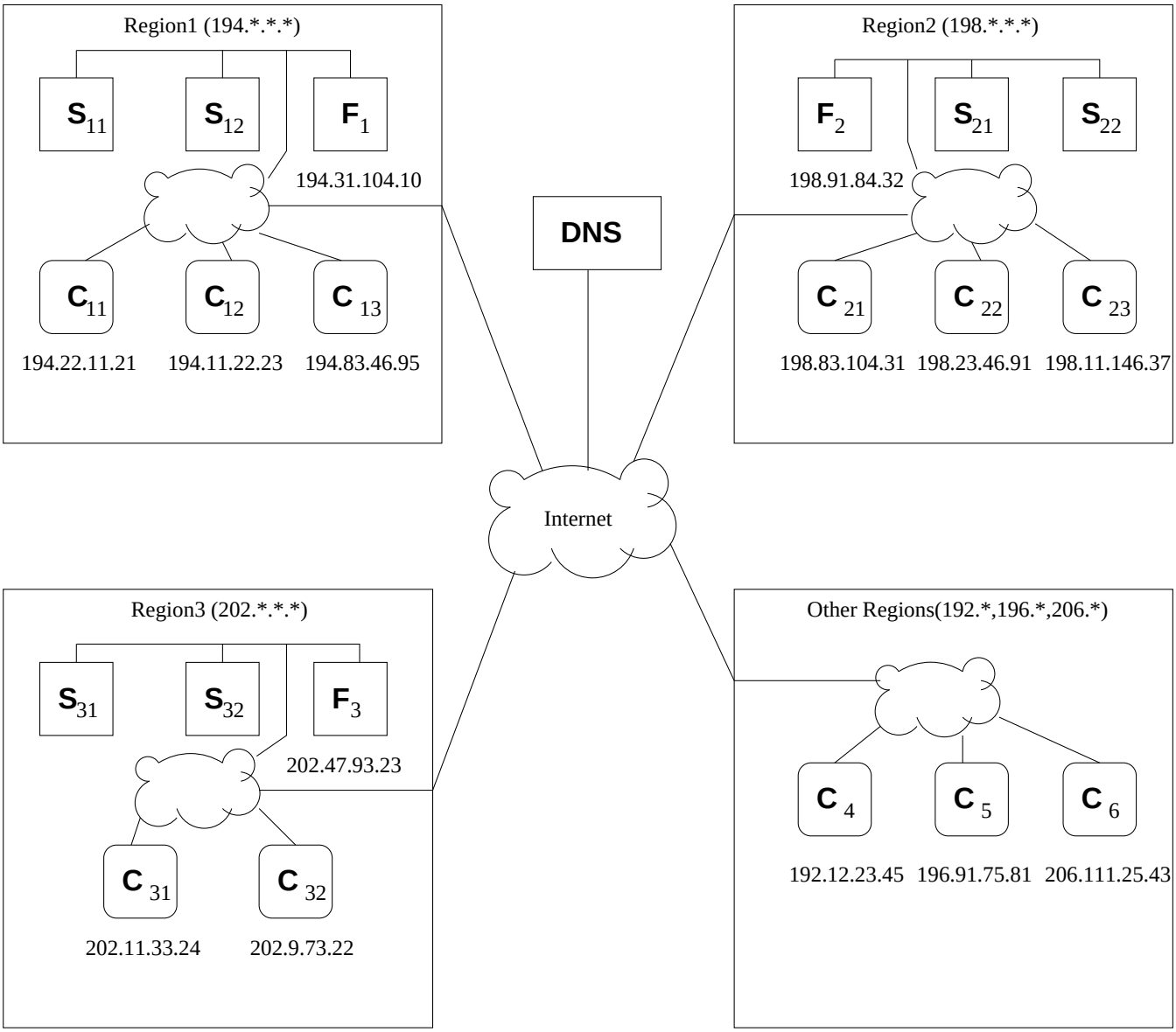


Figure 4.2: Test bed used in Experiments

Chapter 5

Results

We described the setup of test bed used for performing experiment in the last chapter. To generate load and measure performance we have used *Webstone* originally developed by Silicon Graphics and is now maintained by mindcraft.com. This is standard software used to benchmark commercial web servers. Different schemes were tested with everything kept identical except policy for cluster selection at DNS. Webstone software's master process controlling clients was run on one of client PCs.

Experiments were conducted by varying number of client processes from 20 to 120 in steps of 10. Webstone tries to execute roughly equal number of processes at each client. To generate uneven distribution of requests, we wrote the ID of same client machines multiple times in its configuration file. These machines then generated more load than others. When number of client processes were 20,40,..,120, clients in all geographical regions generated almost equal load (per client load i.e. number of processes running were still different). When number of client processes were 30,50,..,110 clients in region1 and region2 were running twice as many client processes as they were running with 10 less client processes (i.e. at 20,40...), while other clients were still running same number of processes, so load was highly uneven.

We have run at least ten iterations of one minute duration each for each data point and taken average of them for plotting. Each Webstone client processes made just single query to DNS before sending requests to servers (clusters) and used resolved mapping for whole testing period of one minute. So due to application

level caching by webstone clients, requests from same client process reached to same cluster for one minute duration regardless of TTL value provided by DNS.

5.1 Architectures emulated on test bed

In our experiments, we were unable to stress web server with heavy load due to limited available RAM (32 MB) and client machines were not able to handle heavy data rate or run large number of webstone processes. Due to packet delay software (which was run as kernel module), when data rate was high, buffering large amount of data for delay period consumed more RAM and generated very high interrupt rate and Linux kernel did not handle the situation gracefully. Even kernel compiled with option "CPU is too slow to handle full bandwidth" did not make systems stable when the data rate was high. Due to these limitations, we could not create the situations when queuing or processing delays at server dominate network delays.

We have emulated four policies for cluster selection at DNS in our test bed and we discuss the results obtained for those policies below:

5.1.1 Round robin selection

In round robin scheme, DNS resolves address of first cluster for first DNS query, of second cluster for second query and so on. After giving addresses of all servers, it starts resolving address of first server again.

Round robin selection policy is very popular DNS scheme. It is used to equally distribute load on multiple servers of same capacity if it is assumed that all the clients generate same number of requests. But in practice, many clients generate very high or very low load thus resulting in load skew.

Average response times as reported by webstone is plotted in Figure 5.1. It is quite clear that there is not much variation in average response time as servers were never bottleneck in service of requests and their service time did not change much.

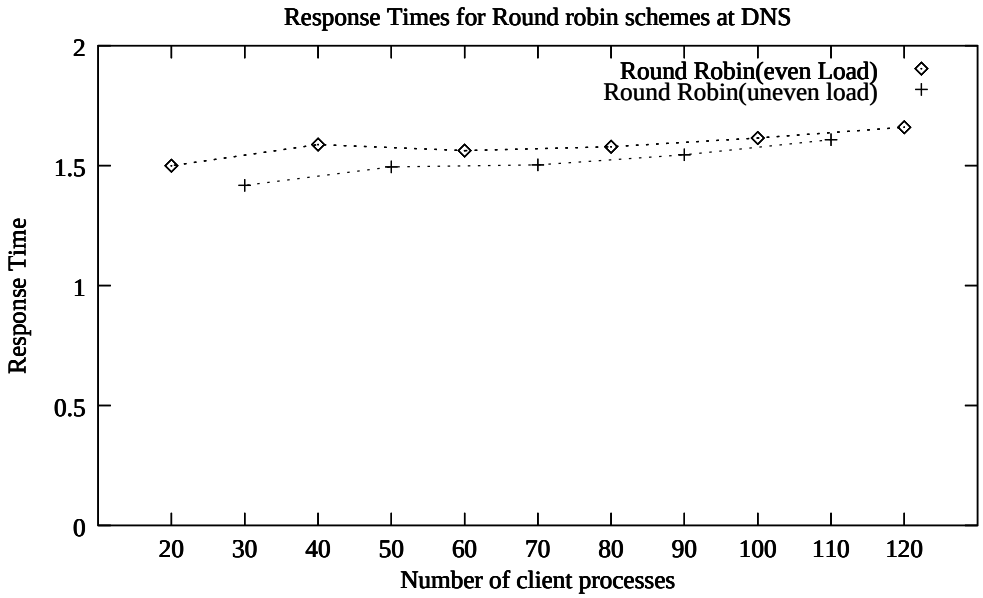


Figure 5.1: Average response time with Round robin policy used at DNS

5.1.2 Random selection

As the name indicates, random policy selects any cluster randomly for each query thus this policy should also resolve IP address of each cluster equal number of time in the long duration. But as opposed to round robin, for very small duration IP address of one cluster may be resolved many times more than that of others. This is the policy (however coupled with shuffling of IP addresses) implemented in BIND.

Average response times reported by webstone is plotted in Figure 5.2. Since selection of server was random, average response time measures also seems to have no fixed pattern.

5.1.3 Weighted capacity selection

In weighted capacity selection, each cluster is assigned either a static weight measured off-line (for example server 2 is twice as powerful as server 1 and 3) or may dynamically report about free capacity of clusters to DNS. DNS returns IP address

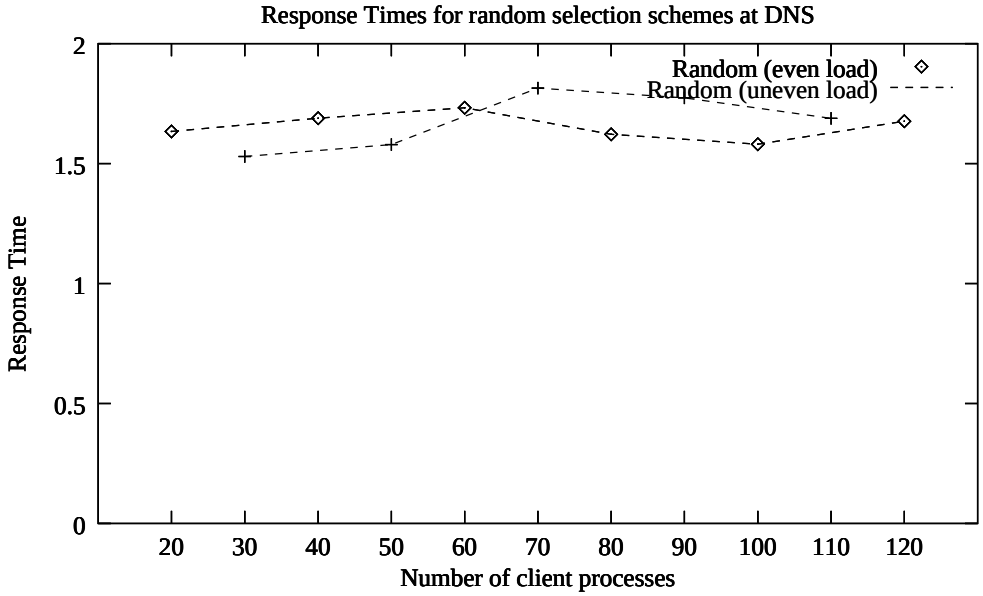


Figure 5.2: Average response time with Random selection policy used at DNS

of particular server in proportion to its weight or free capacity as reported by cluster. We implemented dynamic status reporting based weighted selection. To return IP addresses in proportion to their weight following algorithm is used:

1. Generate running sum of weights associated with each cluster
2. Generate random number between one and sum of weights
3. Return cluster having least running sum of weights and having running sum of weights greater than or equal to the generated random number.

This algorithm is used for servers having different capacity and if used with dynamic capacity reporting, it can deal with load skew due to uneven request rate easily.

Average response times reported by webstone is plotted in Figure 5.3.

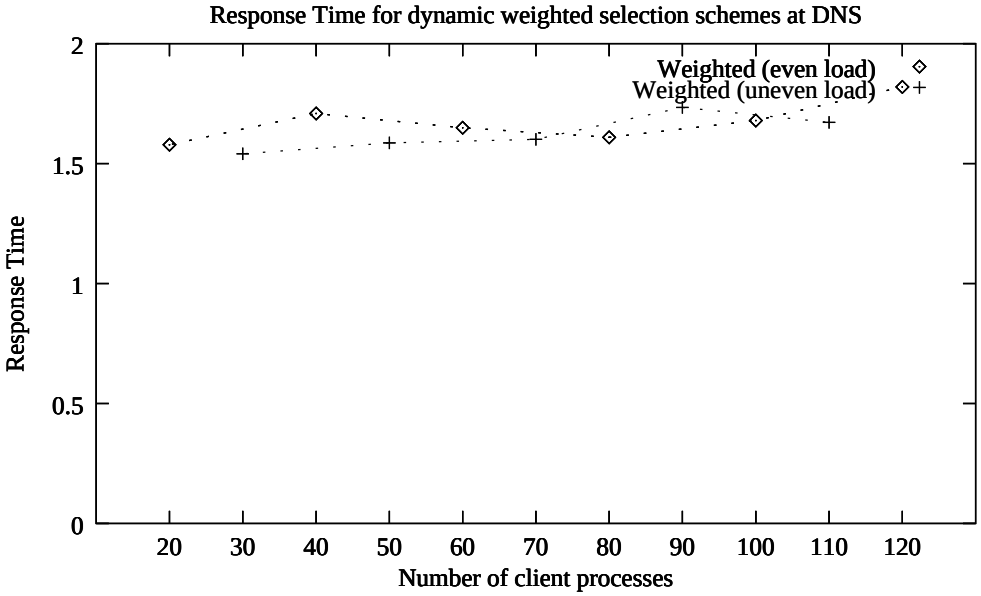


Figure 5.3: Average response time with dynamic Weighted policy used at DNS

5.1.4 Nearest cluster selection

This is the scheme proposed by us. Here, DNS tries to send address of geographically nearest cluster to client if that server is not overloaded. In our scheme, record of clients generating heavy requests (much more than average) is kept, so that these clients do not get IP address of server that is already loaded heavily. So if the request from client comes for the first time or it is not high request rate generating client, DNS gives address of geographically nearest server with enough free capacity to serve requests. This geographical proximity is estimated using IP addresses of cluster and client, for better estimates, local snapshot of whois database may be also queried. In our emulation, we have used high order IP address bits to compare nearness of clients and servers. For giving better performance, we have made policy adaptive. If client generates heavy request rate, its request rate is reported by clusters and we pro-actively request few possibly nearest clusters, having free enough capacity to serve the requests generated from clients, to measure round trip time between them and client. RTT is definitely better but costlier metric to get but this overhead is

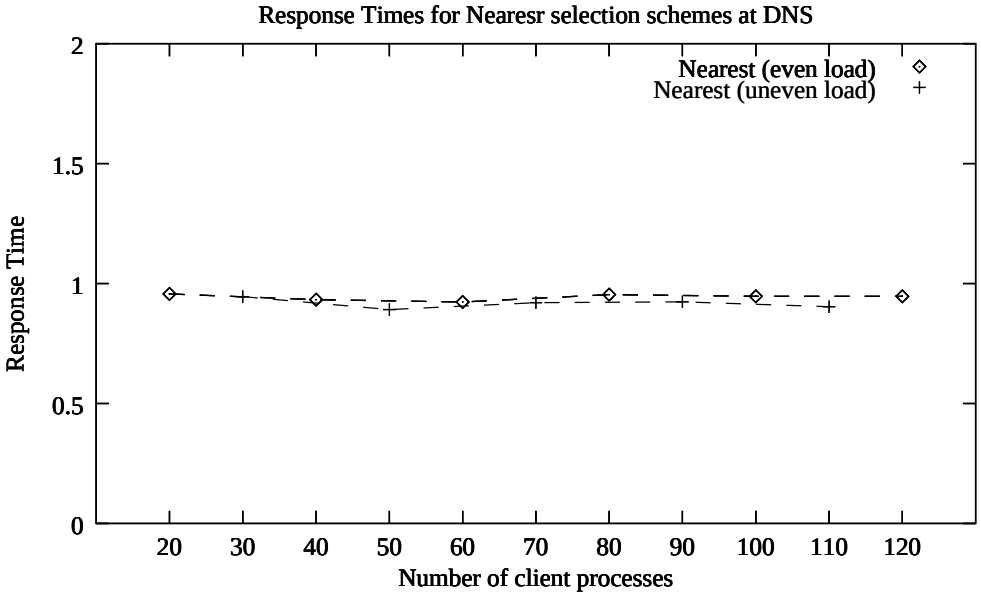


Figure 5.4: Average response time with Nearest server selection policy used at DNS

very small (less than 1% of traffic increase if ping is done to all clients, as reported by Crovella et al [14] in their study). Thus DNS gets better and much more accurate proximity information between clusters and client. Since DNS gives IP address of clusters having enough free capacity, if there is no sudden variation in request pattern of clients, no server should be overloaded in spite of load skew. RTT information is refreshed after refresh time interval. Pseudo code for the algorithm is given in section 3.6.1.

Average response times reported by Webstone is plotted in Figure 5.4. As seen in plot, once again variation is very small but average response time is much better than the other three policies.

5.2 Performance Comparison

We have plotted average response time with different load distribution for different policies in Figure 5.5 and Figure 5.6. As the plots show that under the network

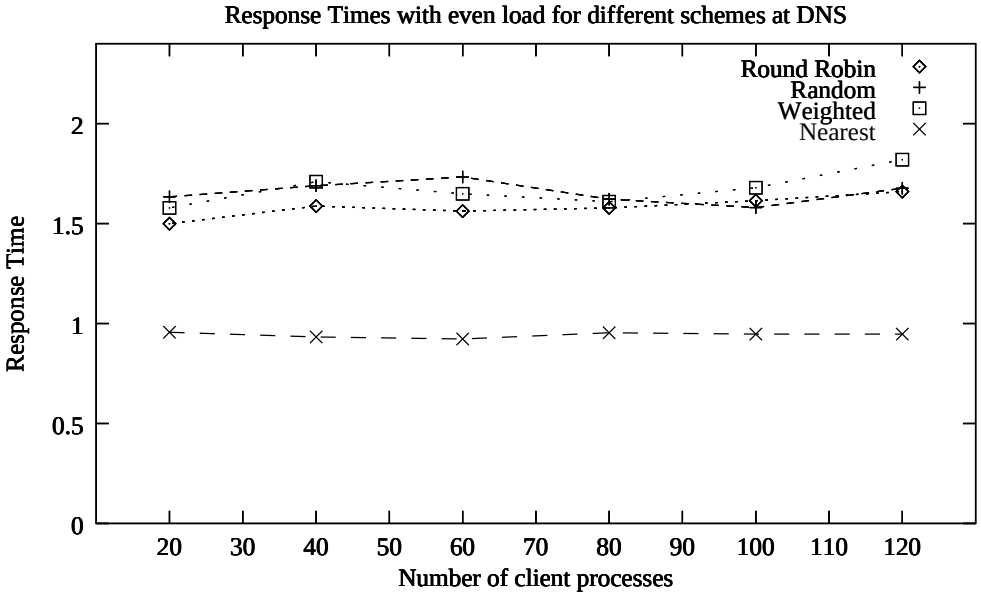


Figure 5.5: Average response time (even load) with different policies used at DNS

conditions assumed by us, our RTT based nearest cluster selection approach outperformed other approaches by a good margin. While other approaches have average response times in range of 1.5 seconds to 1.8 seconds, our approach gave average response time in range of 0.92 second to 0.96 seconds. Thus our results verify that if the links connecting different geographical regions have much higher delay and higher packet losses as compared to links within same geographical region (which is usually the case), we can provide better response time to clients by taking into account the network conditions by using round trip time.

We have also plotted maximum response time for any connection under different policies, we once again see that our policy performs better. These results would be much better if clients in other geographical regions had lesser delays and packet losses with any of nearby cluster (we had set up higher delays and high packet losses with every cluster). The results are plotted in Figure 5.7.

Other two plots, average connection rate (number of connections/sec) and server throughput are shown in Figure 5.8 and Figure 5.9 respectively. The large difference in connection rate and hence higher throughput is attributed to aggressive sequential

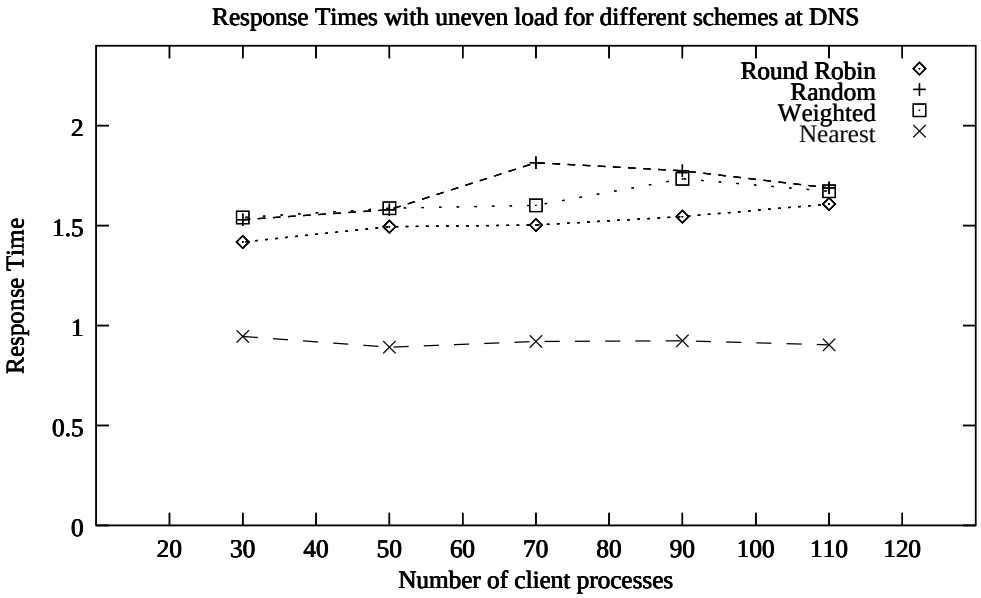


Figure 5.6: Average response time (uneven load) with different policies used at DNS

connection policy used by Webstone software, which tries to send connection requests as fast as possible if earlier requests are serviced quickly. Almost similar response time for varying number of client requests also shows that in our test bed requests were distributed properly by all policies in most cases and servers were not loaded enough.

In our proposed system, more servers and clusters can be added easily without bringing down the system. Our system is also fault tolerant since if any server in the cluster goes down, front node does not receive system state information and does not send any new requests to that server. However, all connections already established with that server are not gracefully handled. Similarly, DNS did not resolve IP address of cluster that went down, clients who were unable to connect to resolved cluster, try other cluster IP addresses and connect to other clusters. This time too, clients having already established connection with that cluster get errors but no new connection afterwards is scheduled to cluster until it comes up again.

In short, we can conclude that our architecture scaled well and our proposed nearest cluster selection approach should give better results if network conditions

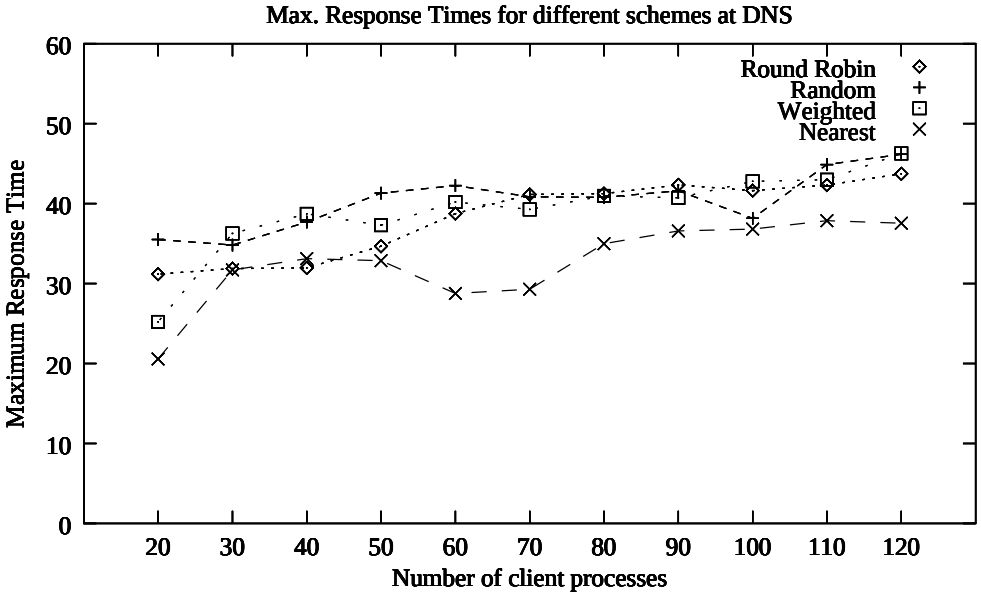


Figure 5.7: Maximum response time with different policies used at DNS

for access within same geographical region are much better than network conditions while accessing clusters in other geographical regions.

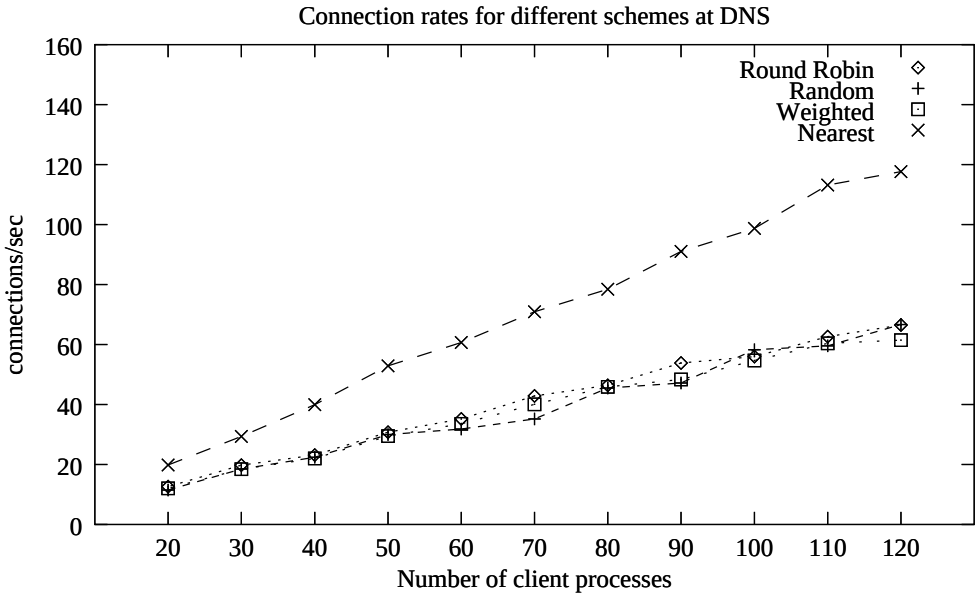


Figure 5.8: Connection rate with different policies used at DNS

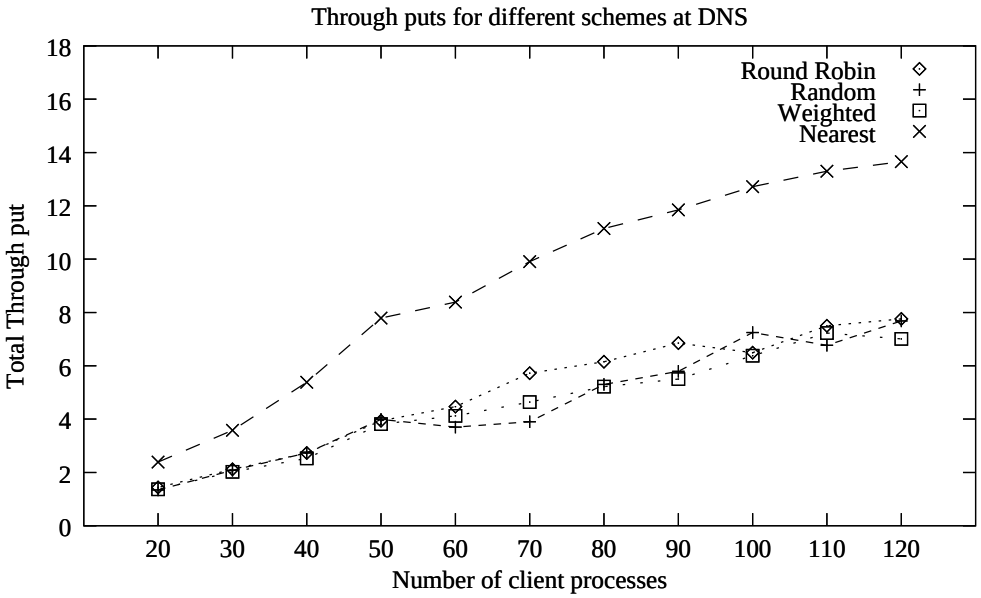


Figure 5.9: Total through put with different policies used at DNS

Chapter 6

Conclusion and Future Extensions

We designed and implemented a test bed for evaluation of load balancing strategies for distributed web server systems. This test bed is quite flexible and new policies can be compared with already existing policies very easily. This test bed will help in understanding trade offs and impact of different parameters on a distributed web server system.

In our thesis, we proposed an adaptive and dynamic policy for server selection and request distribution for a very large website. This DWSS can be deployed with current infrastructure and protocols in use. This architecture is scalable and fault tolerant too. In short, it meets all goals mentioned in design section.

We modified IP packet forwarding method to rewrite only incoming IP packets using shared common IP packets. This can be implemented totally at application layer with divert socket and IP firewalling support, since packets from clients are much shorter, even at application layer there is less overhead as compared to rewriting reply packets which was used in earlier proposed request distribution mechanisms.

From results obtained, we can conclude that our architecture will give better results when clients accessing a particular site are spread in different geographical regions and they are far away from each other. Our architecture is geographically scalable as well as fault tolerant for new incoming requests. Our architecture achieved its main goal of minimizing response time perceived to client.

6.1 Future Extensions

We did not look at the other policies for request distribution within each cluster. Besides it, we assumed that all the servers are capable of serving all the requests. Next step would be consider partial replication on different servers within each cluster and come up with a policy at cluster level to distribute different requests to different servers to get advantage of caching at each server. Next step will be to support different quality of service to different customers or to provide better response time to clients requesting a subset of URL space, for e.g. when they visit pages related to shopping at site.

Our test bed can be further generalized to have switching at different layer of network, we have source code for symmetric and asymmetric splicing too. If support of switching at higher layers is provided, policies that make use of URL or application layer content to select server can also be emulated on our test bed and can be evaluated easily.

After integration of switching at application level, support for partial and dynamic replication can also be added to make it more comprehensive test bed. Once request distribution policies and replication strategies are in same test bed, dynamic replication policies can be explored further which are still not properly understood and explored in distributed web server systems.

Appendix A

Softwares Used

In our test bed we used many third party softwares. In next sections, we briefly describe few mechanisms, software and their use.

A.1 Divert Socket Mechanism

For request distribution at front nodes, we needed some mechanism to capture incoming IP packets for HTTP connections before reaching to TCP layer and rewrite destination address of IP packets and re-inject them back in the network without TCP layer knowing about them. Similarly, we needed mechanism to change destination IP address of incoming IP packets for HTTP connection with local private address, rewrite their destination address and inject back in protocol stack to make TCP layer believe that these packets came for aliased secondary IP address.

We wanted to select destination server address and rewrite all IP packets at application layer at front node. Divert socket provides us exactly same set of features. We used standard firewalling mechanism (that uses high performance packet filtering within Kernel after a setsocketopt call) to divert all IP packet with destination TCP port 80 to divert socket port, where our application read them and modified them. After modification divert sockets provides options to re-inject packets in local TCP/IP stack or on network. At front node, packets are re-injected on network, while at servers packets are allowed to pass up to higher protocol layer in same

machine.

Thus divert socket provided an easy mechanism for request distribution at front node within cluster. However, divert socket requires patching of kernel for divert socket support.

More details about divert sockets can be found at *http://www.anr.mcnc.org/~divert/index.shtml*

A.2 Nistnet

For emulation of WAN characteristics in lab environment, we needed some software to introduce configurable delay and packet losses etc in path of IP packet transfer. Nistnet software allows us to do the same. Nistnet software is now totally modular (with release of version 2.0.10) which does not require patching of kernel, it is installed as loadable kernel module and using its command line interface or GUI based interface, different parameters like delay, packet loss, bandwidth etc. can be set for all incoming IP packets.

More information about nistnet can be found at *http://www.antd.nist.gov/nistnet/*, it is free software from National Institute Of Standards and Technology.

A.3 Webstone

For benchmarking performance of web server system, we used Webstone. It is one of most popular and industry accepted free benchmark program. This software has two parts, a master process and multiple client processes which may be rexeced on remote machines. After establishing trust relationship between clients and master machine, webstone rexecs client process (as specified in test bed configuration file) and client processes generate requests and report back statistics to webstone. Webstone prints performance results like number of connection/sec., connect time, response time, thruput of client and servers, error level, Little's load factor etc.

More information about Webstone can be found at *http://www.mindcraft.com/webstone/*.

Bibliography

[1] Aggarwal, A., and Rabinovich, M. "Performance of Dynamic Replication Schemes for the Internet hosting service". Tech. rep., AT&T Labs., October 1998. http://www.research.att.com/~misha/radar/tm-perf.ps.gz.

[2] Akamai Inc. "How FreeFlow Works". http://www.akamai.com/service/howitworks.html.

[3] Andersen, D., Yang, T., Holmedahl, V., and Ibarra, O. H. "SWEB: Towards a scalable World Wide Web-server on multicomputers". *Proc. of 10th IEEE Int'l Symp. on Parallel Processing, Honolulu* (April 1996), 850–856.

[4] Anderson, E., Patterson, D., and Brewer, E. "The Magicrouter: an application of fast packet interposing". http://cs.berkeley.edu/~eanders/projects/magicrouter/osdi96-mr-submission.ps.

[5] Arlitt, M. F., and Williamson, C. L. "Internet Web Servers: Workload Characterization and Performance Implications". *IEEE/ACM Transactions on Networking, Vol. 5, No. 5* (October 1997), 631–644.

[6] Baentsh, M., Baum, L., and Molter, G. "Enhancing the Web's Infrastructure: From Caching to Replication". *Internet Computing Vol. 1. No. 2* (March-April 1997), 18–27.

[7] Beck, M., and Moore, T. "The Internet-2 Distributed Storage Infrastructure project: An architecture for Internet content channels". *3rd Int'l WWW Caching Workshop, Manchester, UK* (June 1998). http://wwwcache.ja.net/events/workshop/18/mbeck2.html.

[8] C. Partidge, T. M. . W. M. "RFC 1546: Host anycasting service".

[9] Cardelini, V., Colajanni, M., and Yu, P. S. "Dynamic load balancing on web server systems". *IEEE Internet Computing, vol 3, no 3* (May-June 1999), 28–39.

[10] CASAVANT, T. L., and KUHL, J. G. "A Taxonomy of Scheduling in general-purpose Distributed Computing System". *IEEE Transactions on Software Engineering, Vol. 14, No. 2* (February 1988), 141–153.

[11] Cisco Systems Inc. "Distributed Director White Paper". http://www.cisco.com/warp/public/cc/cisco/mkt/scale/distr/tech/d_wp.htm.

[12] Colajanni, M., Yu, P. S., and Cardelini, V. "Dynamic load balancing on geographically distributed heterogenous web servers". *IEEE 18th Int'l Conference on Distributed computing systems* (May 1998), 295–302.

[13] The World Wide Web Consortium "The Common Logfile Format". http://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format.

[14] Crovella, M. E., and Carter, R. L. "Dynamic server selection in the Internet". *Proceedings of the 3rd. IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95)* (June 1995). http://www.cs.bu.edu/faculty/crovella/paper-archive/hpcs95/paper-final.ps.

[15] D. Mosedale, W. F., and McCool, R. "Lessons learned administering Netscape's site". *Internet Computing Vol. 1 No. 2* (March-April 1997), 28–35.

[16] Damani, O., Chung, P., and Kintala, C. "ONE-IP: Techniques for hosting a service on a cluster of machines". *Proceedings of 41st IEEE Computing Society Int'l Conference* (February 1996), 85–92.

[17] Ellen W. Zegura, Mostafa H. Ammar, Z. F., and Bhattacharjee, S. "Application-Layer Anycasting: A Server Selection Architecture and use in a Replicated Service". *IEEE/ACM Transactions on Networking, Vol. 8, No. 4* (August 2000), 455–466.

[18] G.D.H. Hunt, G.S. Goldzsmit, R. K., and Mukherjee, R. "Network Dispatcher: A connection router for scalable internet services". *Proceedings of 7th Int'l World Wide Web Conference* (April 1998).

[19] Gerich, E. "RFC 1466 - Guidelines for Management of IP address space".

[20] Guyton, J., and Schwartz, M. "Locating nearby copies of replicated Internet servers". *Proceedings of SIGCOMM'95, Vol. 25, No. 4* (October 1995), 288–298.

[21] Gwertzman, J., and Seltzer, M. "The case for geographical push-caching". *Proceedings of 1995 Workshop on Hot Topics in Operating System* (1995).

[22] Jacobson, V. "A Tool to infer characteristics of Internet paths.", April 1997. ftp://ftp.ee.lbl.gov/pathchar/.

[23] Kelvin Lai, M. B. "Measuring Bandwidth'. *Proceedings of IEEE INFOCOMM'99, NY* (March 1999).

[24] Kwan, T. T., McGrath, R. E., and Reed, D. A. "NCSA's World Wide Web server: Design and performance". *IEEE Computer, no. 11* (November 1995), 68–74.

[25] Mehmet Sayal, Yuri Breitbart, P. S., and Vingralek, R. "Selection Algorithms for Replicated Web Servers". *Proceedings of the Workshop on Internet Server Performance* (1998). http://www.cs.wisc.edu/~cao/WISP98/final-versions/mehmet.ps.

[26] Mockapetris, P. " RFC 1034 : Domain Names - Concepts and Facilities", November 1987.

[27] Mockapetris, P. "RFC 1035 : Domain Names - Implementation and Specification", November 1987.

[28] Paxon, V. "End-to-End Routing Behaviour in the Internet". *IEEE/ACM Transactions on Networking, Vol. 5, No. 5* (October 1997), 601–615.

[29] R. Luling, B. M., and Ramme, F. "A study on dynamic load balancing algorithms". Tech. rep., Paderborn Center for Parallel Computing, University of Paderborn, Germany, June 1992.

[30] Trent, G., and Sake, M. "WebSTONE: The First Generation in HTTP Server Benchmarking", February 1995. http://www.mindcraft.com/webstone/paper.html.

[31] Yoshilakawa, C., Chun, B., and Eastham, P. "Using smart clients to build scalable services". *Proceedings of Usenix 1997* (January 1997).

[32] Zongming Fei, Samrat Bhattacharjee, E. W. Z., and Ammar, M. "A Novel Server Selection Technique for improving the Response Time of a Replicated Service". *IEEE INFOCOMM '98 Conference* (1998). http://www.cc.gatech.edu/fac/Ellen.Zegura/papers/alas-inf98.ps.gz.