Load Balancing a Cluster of Web Servers †

Using Distributed Packet Rewriting

Luis Aversa Azer Bestavros
Laversa@cs.bu.edu Bestavros@cs.bu.edu

Computer Science Department
Boston University

Abstract

In this paper, we present and evaluate an implementation of a prototype scalable web server. The prototype consists of a load-balanced cluster of hosts that collectively accept and service TCP connections. The host IP addresses are advertised using the Round Robin DNS (RR-DNS) technique, allowing any host to receive requests from any client. Once a client attempts to establish a TCP connection with one of the hosts, a decision is made as to whether or not the connection should be redirected to a different host---namely, the host with the lowest number of established connections. We use the low-overhead Distributed Packet Rewriting (DPR) technique [Bestavros, Crovella, Liu, and Martin 1998] to redirect TCP connections. In our prototype, each host keeps information about the remaining hosts in the system. Load information is maintained using periodic multicast amongst the cluster hosts. Performance measurements suggest that our prototype outperforms both pure RR-DNS and the stateless DPR solutions.

1. Introduction

The phenomenal growth of the WWW is imposing considerable strain on Internet resources and Web servers, prompting numerous concerns about the Web's continued viability. The success of high-performance Web servers in alleviating these performance problems is ultimately limited, unless Web services are designed to be inherently scalable.

To construct scalable Web servers, system builders are increasingly turning to distributed designs. An important challenge that arises in distributed Web servers is the need to direct incoming connections to individual hosts. Previous methods for connection routing have employed a centralized node (termed a TCP router) that acts as a switchboard, directing incoming requests to backend hosts. Under this architecture, a single machine whose IP address is published through DNS takes on the responsibility of balancing the load across the cluster [for examples, see Dias et al 1996 and Cisco Local Director 1997]. This centralized approach is not inherently scalable because it does not take into account the fact that the TCP router becomes a bottleneck at high loads. A proposed alternative to this centralized approach is Distributed Packet Rewriting [see Bestavros et al 1998] (DPR). DPR follows the same idea of distributing requests across a number of web servers to handle high loads of web traffic. The major difference between DPR and TCP routing lies in the manner in which IP addresses are published. DPR uses Round-Robin DNS to publish individual addresses of all machines in the cluster of web servers, thereby distributing the responsibility of re-routing requests to each machine.

In this paper, we demonstrate the feasibility of this decentralized approach through the presentation and performance evaluation of a prototype implementation of a DPR-based distributed server architecture.

---

† This work has been partially funded by NSF research grant CCR-9706685.

2. Related Work

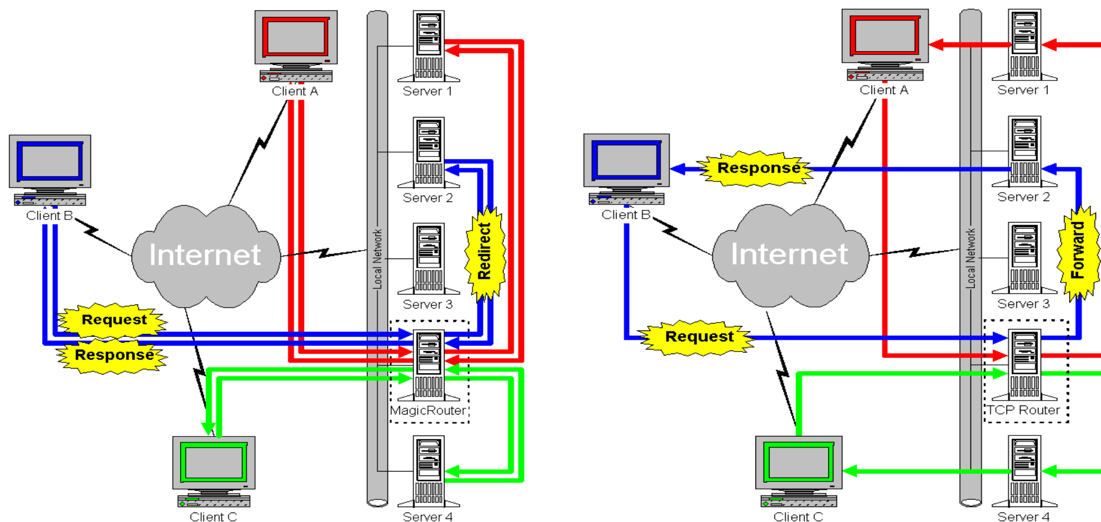
Early work on distribution and assignment of incoming connections across a cluster of servers [see the works of Katz et al 1994, and Mogul 1995] has relied on Round-Robin DNS (RR-DNS) to distribute incoming connections across a cluster of servers. This is done by providing a mapping from a single host name to multiple IP addresses. Due to DNS protocol intricacies (e.g. DNS caching and invalidation), RR-DNS was found to be of limited value for the purposes of load balancing and fault tolerance of scalable Web server clusters. The research described in [Mogul 1995 and Dias et al 1996] quantifies these limitations.

Centralized Connection Routing:

Rather than delegating to DNS the responsibility of distributing requests to individual servers in a cluster, several research groups have suggested the use of a local ``router" to perform this function. For example, the NOW project at Berkeley has developed the MagicRouter [Anderson, Patterson, and Brewer 1996], which is a packet-filter-based approach [See Mogul, Rashid, and Accetta 1987] to distributing network packets in a cluster. As illustrated in Figure 1 (a), the MagicRouter acts as a switchboard that distributes requests for Web service to the individual nodes in the cluster. To do so requires that packets from a client be forwarded (or ``rewritten") by the MagicRouter to the individual server chosen to service the client's TCP connection. Also, it requires that packets from the server be ``rewritten" by the MagicRouter on their way back to the client. This *packet rewriting* mechanism gives the illusion of a ``high-performance" Web Server, which in reality consists of a router and a cluster of servers. The emphasis of the MagicRouter work is on reducing packet processing time through ``Fast Packet Interposing"---but not on the issue of balancing load. Other solutions based on similar architectures include the Local Director by Cisco [See Cisco 1997] and the Interactive Network Dispatcher by IBM [See IBM 1997].

An architecture slightly different from that of the MagicRouter is described in [Dias et al 1996], in which a ``TCP Router" acts as a front-end that forwards requests for Web service to the individual back-end servers of the cluster. Two features of the TCP Router differentiate it from the MagicRouter solution mentioned above. First, as illustrated in Figure 1 (b), rewriting packets from servers to clients is eliminated. This is particularly important when serving large volumes of data. To allow for the elimination of packet rewriting from server hosts to clients requires modifying the server host kernels, which is not needed under the MagicRouter solution. Second, the TCP Router assigns connections to servers based on the state of these servers. This means that the TCP Router must keep track of connection assignments.

The architecture presented in [Law, Nandy, and Chapman 1997] uses a TCP-based switching mechanism to implement a distributed proxy server. The motivation for this work is to address the performance limitations of *client-side* caching proxies by allowing a number of servers to act as a single proxy for clients of an institutional network. Their architecture uses a centralized dispatcher (a Depot) to distribute client requests to one of the servers in the cluster representing the proxy. The function of the Depot is similar to that of the MagicRouter. However, due to the caching functionality of the distributed proxy, additional issues are addressed---mostly related to the maintenance of cache consistency amongst all servers in the cluster.



(a) TCP Router with 2-way packet rewriting (b) TCP Router with 1-way packet rewriting

Figure 1: Centralized Connection Routing Architectures

Distributed Connection Routing:

All of the above connection routing (also known as Layer 4 Switching) techniques have employed a centralized node which handles all incoming requests. In contrast, the Distributed Packet Rewriting technique presented in [Bestavros, Crovella, Liu, and Martin 1998] (DPR), also called Distributed Routing in [CNT Inc. 1999], *distributes* that functionality. As illustrated in Figure 2, using DPR, all hosts of the distributed system participate in connection routing. This distributed approach promises better scalability and fault-tolerance than the predominant use of centralized, special-purpose connection routers.

DPR is an IP level mechanism that equips a server with the ability to redirect an incoming connection to a different server in the cluster based on the very first packet (SYN packet) received from the client. This implies that the redirection decision (i.e. which server ought to be chosen for redirection) can only rely on the information included in the SYN packet---namely, src/dst IP addresses and src/dst port numbers---as well as on cluster state information---e.g., relative load on the different servers in the cluster. Using this information, a DPR-enabled server either forwards a connection to a different server, or lets it percolate up its network stack to the application layer. There are two versions of DPR, stateless and stateful. Stateless DPR does not require any information different from what can be found in the headers of each packet in a connection. Thus, forwarding is done independently on a packet by packet basis according to a hash function. Stateful DPR keeps a table of translations, which is used to determine where to forward packets of a given connection (based on a choice made initially upon receipt of the connection's SYN packet).

In [Bestavros, Crovella, Liu, and Martin 1998], DPR was tested using a randomizing re-routing algorithm (to determine whether or not to forward packets or serve them locally). Based on a hash function that was applied to the source port number of the TCP packet, the decision was made. This approach is entirely stateless – it does not rely on feedback from other machines regarding current load in order to make the determination of whether to forward a packet. In this paper, we argue and show that using a stateful

approach (using accurate load estimation on the machines in the cluster) to distribute packets will achieve better throughput and a faster mean response time to the client.

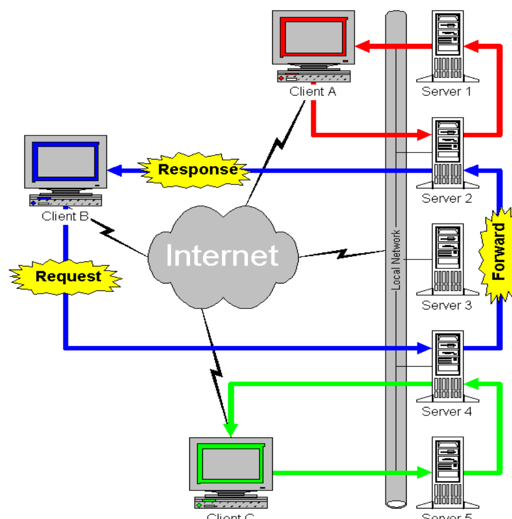


Figure 2: Distributed Connection Routing Architecture

3. DPR Prototype Implementation

In our implementation of DPR, each machine within the cluster provides web service, along with the ability to re-route requests to other machines through packet rewriting. The IP addresses of all the machines in the cluster are advertised through RR-DNS, allowing any of the machines to receive requests. Such requests can be either served locally or re-routed to another machine. In the latter case, the responsibility of serving the request will be transferred to another machine, which will respond directly to the client.

3.1. Overview

To illustrate this packet rewriting scheme, assume that Server 4 (see Figure 2) receives the original request from Client B. Furthermore, assume that it is decided (based on load information, or some other criteria) that Server 4 should not serve this incoming request, but rather to route this request to Server 2. To do so, Server 4 acts as a router, "rewriting" the packets from Client B to Server 2. Server 2 serves the request to Client B, using the IP address of Server 4 as the source IP, effectively masking the process of re-routing the request from Server 4 to Server 2. Client B will continue to send packets to Server 4, unaware of the fact that the request had been forwarded to Server 2. Notice that while Server 4 is acting as a router for Client B's request, it may actually be serving other requests (e.g. Client C's request, which has been re-routed to it from Server 5).

In our implementation, it is necessary for the machines within the cluster to distinguish between packets that have been re-routed and packets that come directly from the client. Furthermore, if Server 4 re-routes a request to Server 2, then Server 4 must let Server 2 know Client B's IP address in order for Server 2 to

respond to Client B's request properly. To address both of these issues, we employ IP-IP encapsulation [See IETF RFC 2003].

Using IP-IP encapsulation, Server 4 encapsulates the original packet received from client B inside another IP packet, which is then re-routed to Server 2. Server 2 is now able to deduce that the packet was re-routed by Server 2, and can respond to client B accordingly. It can also find the source IP of Client B within the encapsulated packet in order to complete the request.

To enable a stateful routing of requests using DPR, each machine keeps an updated list of all other machines within the cluster, with information such as their IP addresses and current load. Hosts intermittently broadcast their load to the other machines (using multicast UDP packets). This information is used by a server to determine whether an incoming request should be re-routed or whether it should be served locally. Also, each machine keeps routing tables with information about redirected connections.

The particular distributed load-balancing algorithm we use works as follows. When a new request (i.e. the SYN packet of a TCP connection) is received by Server 4 from client B, Server 4 first examines its own load. If the load is under a certain threshold value `MaxLoad`, then Server 4 will serve the request locally. If not, Server 4 will create a new entry in its routing tables and will forward the request (i.e. the SYN packet of the TCP connection) to one of the other servers in the cluster. Subsequent packets from this connection are routed according the information in the routing table. This threshold value `MaxLoad` can be adjusted according to certain factors such as CPU speed, memory, etc.

We used two different approaches to select the server to which a request is re-routed. The first approach is deterministic, whereby the server with the lowest load is selected. The second approach is probabilistic, whereby the probability of selecting a server is inversely proportional to the load on that server. The advantage of this latter approach is that it avoids possible oscillations (whereby all requests in a short timeframe are re-routed to the server with the lowest advertised load, potentially overloading such a server).

We used a number of different metrics to estimate the "load" on the servers—namely,

(1) the total number of open TCP connections each machine in the cluster has at any given moment,
(2) the CPU utilization of each machine in the cluster,
(3) the number of redirected TCP connections by each machine in the cluster, and
(4) the number of active sockets of each machine in the cluster.

In addition, we have experimented with various functions that combined the above four metrics using different weights and functions.

Our implementation of stateful distributed connection routing was done under linux 2.0.28. It consisted of two main components: one in kernel space and one in user space. The first component required the design of a very fast mechanism to search, insert, delete and update real-time data for routing purposes. This mechanism was implemented entirely in the kernel using multiple hash tables and linked list. The second component was to design a mechanism to store the information regarding other machines' current loads and update such information periodically (e.g. every second). A sorted linked list, three user processes and new systems calls were needed for the implementation of this component.

3.2. Implementation of Routing Functionality in Kernel Space

When a machine receives an IP packet, the kernel calls the function **`ip_receive()`**. Some modifications were made to this function to be able to redirect connections. In this function, the IP packet is examined. If it contains a TCP packet and the TCP destination port is 80 (or whatever other port the web server is running on), we know that such a TCP connection it is an HTTP connection and is coming

directly from the client. If the TCP packet contains a SYN, then we know that a new connection is being requested. A decision has to be made, to serve it locally or to forward it. As eluded to earlier, this decision is based on the load table and the current load of the machine. If the machine is under the threshold value or the current load of the machine is the lowest compared to the other machines then the request is served locally and no routing tables are updated. If the current load is above the threshold value and the lowest load correspond to another machine then the routing tables are updated and the packet is forwarded to some other server (using either the deterministic or probabilistic approaches we discussed earlier). If the TCP packet is not a SYN then, we look up in the routing tables and if the connection has been redirected, then the packet is forwarded. If the IP packet contains an IP-IP packet and the unused bit of the fragment offset is set to 1, we know that it is a packet that has been redirected and that we have to serve it. We unpack the IP-IP packet and send the TCP packet to the TCP layer to be processed. Instead of utilizing the unused bit of the fragment offset, we could check if the source IP address correspond to the servers participating in the DPR to detect redirected connections.

3.2. Implementation of Cluster Load Information Functionality in User Space

The mechanism to maintain an accurate view of the load on the various servers in the cluster was implemented with three user processes and seven new system calls. One process is in charge of broadcasting the local server's own load periodically (in our experiments, we set the period to 1 second). To get local load information, this process makes a system call to obtain the appropriate value of the load (namely: CPU utilization, number of open TCP connections, number of active sockets, and number of rerouted connections). A second process is in charge of waiting for the load of the other servers that are participation in the DPR protocol to be multicast. Every time a new value is received, the process makes a system call to update the sorted linked list maintained in the kernel. The third process is in charge of cleaning up of the load and the routing tables. If no load packet is received from one machine for a certain number of second, then the entry of this machine in the load table is deleted to avoid redirecting connection to a machine that is not running (e.g. due to a failure or a periodic maintenance shutdown).

Using IP-IP to redirect connections allows us to have servers in different networks. We only need to tell the process in charge of broadcasting the load the networks that participate in DPR. If more than one network have servers participating in DPR, this process will broadcast the load packet not only to the local network but also to all other networks participating in this protocol. The identity of all participating networks is captured from a configuration file upon the initialization of this process.

4. Performance Evaluation

In this section we present the results of our performance evaluation of our prototype implementation of the stateful distributed connection routing architecture.

4.1. Experimental Setup

In order to evaluate the performance and the load distribution of the implementation, we used a URL request generator tool called SURGE [see Barford and Crovella 1998] (Scalable URL Reference Generator) to create a realistic web workload. Surge is a tool developed as part of the Commonwealth project [See Commonwealth Project, 1997] that attempts to accurately mimic a fixed population of users accessing a Web server. It adheres to six empirically measured statistical properties of typical client requests, including request size distribution and inter-arrival time distribution. Surge adopts a closed system model (workload is generated by a fixed population of users, which alternate between making requests and lying idle).

SURGE was run in each client machine with the following parameters: five client sub-processes with 50 threads each for 200 seconds. We ran SURGE from six machines that were generating requests to three Pentium-class web servers (266 Mhz, 128MB, 100Mbps Ethernet) running apache. These servers are named: Brookline, Baystate and Buick. Four SURGE clients were generating requests to Buick, one to Brookline and one to Baystate as shown in Figure 3. This *uneven* assignment of SURGE clients to servers results in a heavy load being offered to one of the machines (namely Buick). As documented in previous studies [See Mogul 1995 and Dias et al 1996], this is typically what happens when round-robin DNS is used to map a domain name to a set of IP addresses.[1]

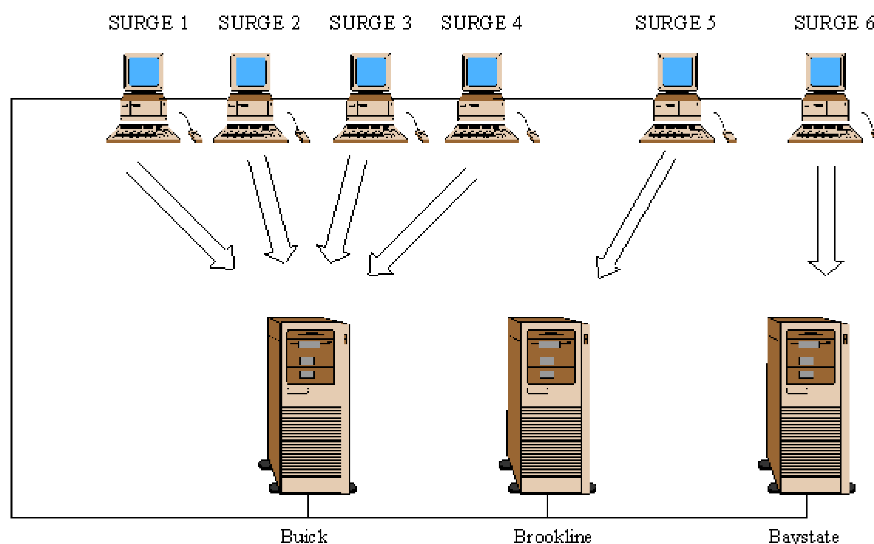


Figure 3: Experimental Setup Used in Performance Evaluation of Prototype Implementation

We show in the next section the behavior of the cluster applying different algorithms to balance the load. First, we ran the test using not load balancing at all, second we used the random load balancing and third, we used the TCP load balancing algorithm explained in the implementation section.

4.2. Test Scenarios and Metrics

Three scenarios were tested. In the first (termed “No Load Balancing”), we ran the system with the DPR functionality turned off. This scenario represents RR-DNS solutions for assigning client requests to cluster hosts as described in [Mogul 1995]. In the second (termed “Random Load Balancing”), we ran the system with DPR functionality enabled, but with a stateless (random) rerouting policy. This scenario is akin to that used in [Bestavros, Crovella, Liu, and Martin 1998]. In the third (termed “TCP Load Balancing”), we ran the system with DPR functionality enabled and with a stateful re-routing policy that uses the total number of TCP connections to a server as a measure of load.[2]

---

[1] In [Mogul 1995] the imbalance caused by RR-DNS was empirically characterized for a cluster or 3 servers. The measurements suggest that over 25% of the time, the most-loaded server sustained almost 60% of the of the total load in the system. In [Dias et al 1996], it was shown that the peak load on nodes of a cluster as a result of RR-DNS can be up to 40% higher than the mean load on all nodes and that this load imbalance is independent of the number of servers in the cluster.

[2] We have also evaluated a host of other policies using other load metrics (as described earlier in this paper). Our findings suggest that using the number of concurrent TCP connections as a measure of load was *consistently* either the best policy or within 5% of the best policy. Thus, in the remainder of this section, we restrict our presentation of performance results to the performance of TCP Load Balancing.

To evaluate these three approaches, we measured the mean and variance of the transfer delay of documents (as measured by SURGE clients) as well as the total number of requests served and the rate of service (or throughput).

4.3. Test Results

Table 1 shows the metrics we obtained for each of the tested scenarios. Clearly, TCP load balancing outperforms the other scenarios in both the mean transfer delay and the number of requests served per second.

Policy Used	Transfer Delay		Requests Served	
	Mean	Variance	Total	Rate / sec
No Load Balancing	0.918775	15.240970	96,726.00	496.03
Random Load Balancing	0.372362	0.813577	123,798.00	634.86
TCP Load Balancing	0.263267	0.859490	129,278.00	662.96

Table 1: Performance of Various Policies

The three graphs shown in Figures 4, 5, and 6 capture the behavior of the cluster under the three scenarios tested. They show how many connections each machine serves per second. When we use no load balancing, we can see that Buick served the majority of requests. When we use Random load balancing or TCP load balancing we can see that the three servers are serving approximately the same number of connections per second leading to a *better* response time and throughput.

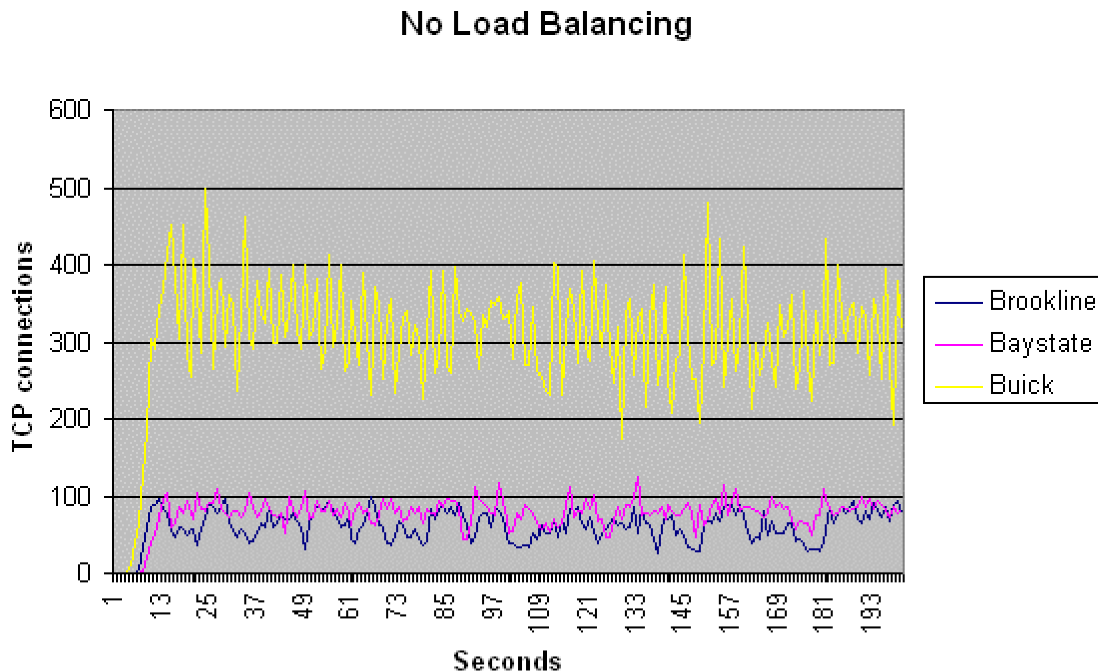


Figure 4: Total number of TCP connections served by each server with no Load Balancing

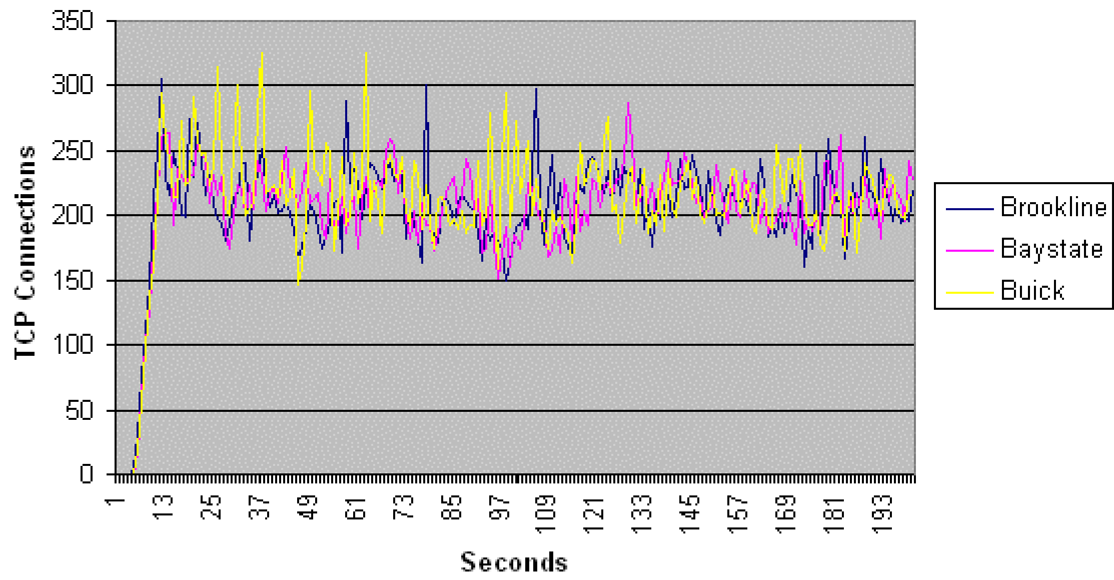


Figure 5: Total number of TCP connections served by each server with Random Load Balancing

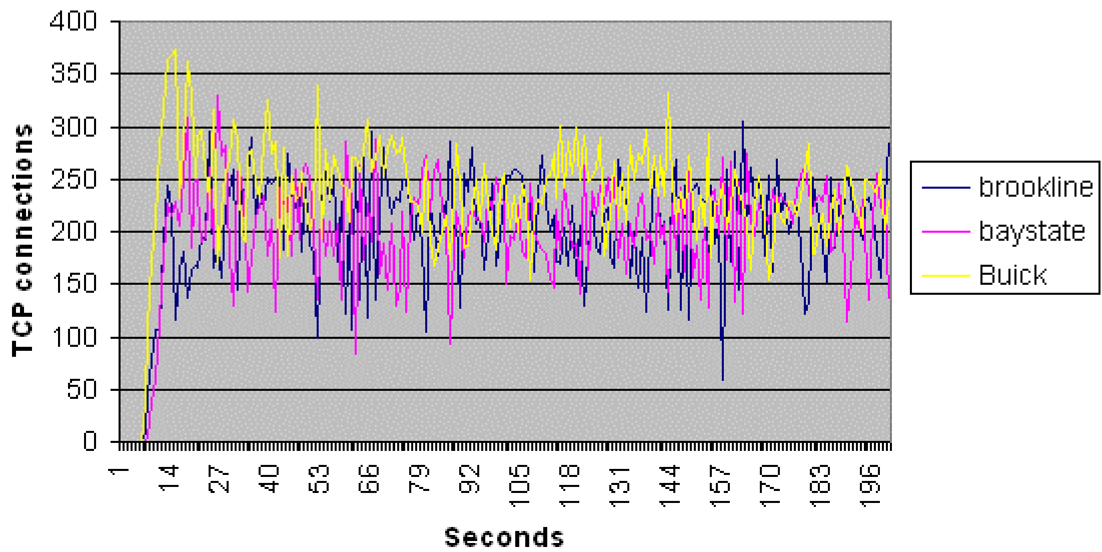


Figure 6: Total number of TCP connections served by each server with TCP Load Balancing

The instantaneous measurements shown in Figures 4, 5, and 6 are aggregated in a histogram to show the load variability under each of the three policies (aggregated for all servers). Figures 7, 8, and 9 show these results. The X-axis represents a range of load (measured in terms of concurrent open TCP connections) and the Y-axis represents the number of observations that corresponded to that load in our tests. A distribution with a "wider" spread is indicative of an inferior load balancing policy, whereas a steeper distribution is indicative of a more efficient policy. Clearly, TCP Load Balancing achieves the minimum spread and hence provides the best load balancing performance. Table 2 summarizes these results by showing the mean, 10th percentile, 90th percentile, and the load imbalance index for each of these policies.

Method Used	**Observed Load in Terms of # of concurrent TCP Connections**			**Imbalance Index**
	μ	**10th Percentile**	**90th Percentile**	$1 + (\Delta/2) / \mu$
No Load Balancing	462	372	557	1.20
Random Load Balancing	643	573	722	1.16
TCP Load Balancing	660	605	717	1.08

Table 2: Relative performance of the three tested policies

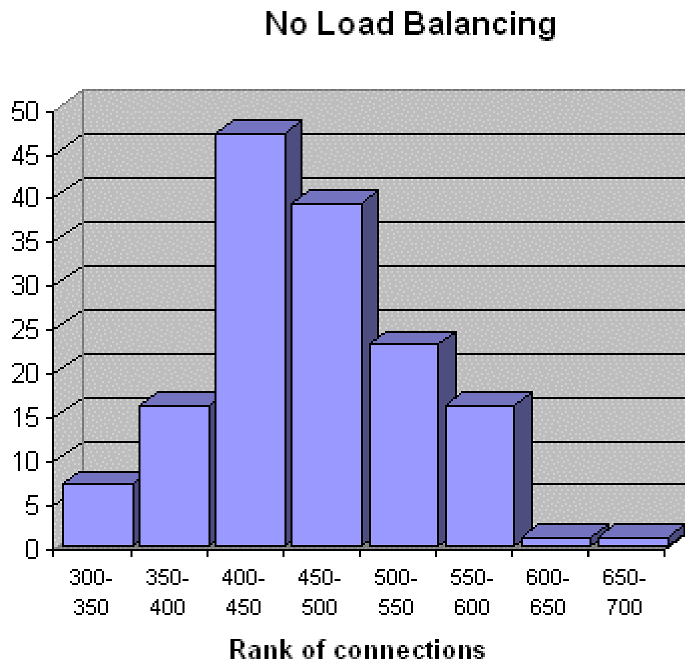


Figure 7: Observed Load Conditions under a No Load Balancing policy

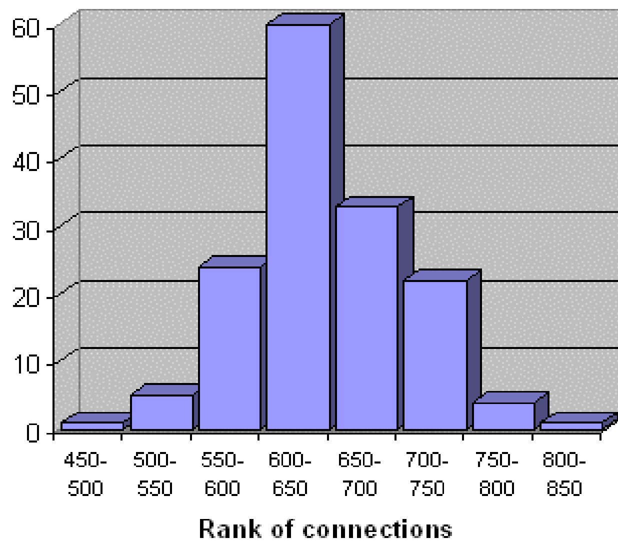


Figure 8: Observed Load Conditions under a Random Load Balancing policy

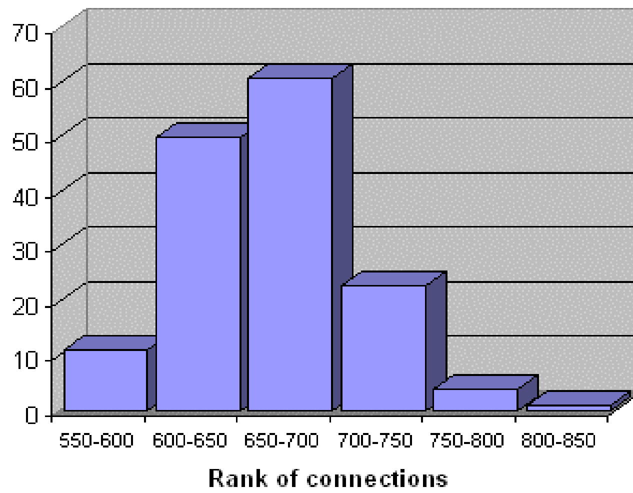


Figure 9: Observed Load Conditions under a TCP Load Balancing policy

5. Conclusion

In this paper we have presented and experimentally evaluated a prototype implementation of distributed connection routing using the DPR technique described in [Bestavros, Crovella, Liu, and Martin]. DPR allows routing connections in a distributed server without employing any centralized resource. Instead of using a distinguished node to route connections to their destinations, as in previous systems, DPR involves *all* the hosts of the distributed system in connection routing. In the work presented in [Bestavros, Crovella, Liu, and Martin], the main idea behind DPR was presented and its scalability was asserted. In this paper we have focussed on a more *realistic* prototype implementation of DPR, using IP-IP encapsulation and a stateful load-cognizant approach to load balancing.

One concern about such an approach is that the addition of connection routing to the responsibilities of the hosts in a server may overburden them with an unacceptable amount of additional work. We have shown that in our implementations, this is not the case. Furthermore, we believe that architectural trends will increasingly favor the co-location of packet routing functions with other system functions in individual hosts. This is because I/O interface hardware, and network interface cards in particular, are rapidly increasing in sophistication. The Intelligent I/O initiative I^2O [See I^2O SIG] is in fact standardizing hardware and software interfaces for the use of highly intelligent I/O cards in general purpose computing systems. As these trends accelerate, approaches like DPR will become even more attractive.

The functions of DPR do not completely replace those of a centralized connection router such as the Network Dispatcher [See IBM 1997] or Local Director [See Cisco 1997]. Such connection routers present a single IP address while performing packet rewriting, load balancing, and (potentially) network gateway functions (that is, IP routing, firewall functionality). DPR does not present a single IP address, and does not perform network gateway functions. However, we have shown that simple RR-DNS is sufficient for providing the illusion of a single IP address, and standard routers are sufficient (and preferable) for providing gateway functions.

The benefits that DPR presents over centralized approaches are considerable: the amount of routing power in the system scales with the number of nodes, and the system is not completely disabled by the failure of any one node. DPR also has special value for small-scale systems. For example, consider the case in which a Web server needs to grow in capacity from one host to two. Under a centralized approach, two additional hosts must be purchased: the new host *plus* a connection router, even though most of the capacity of the connection router will be unused. DPR allows more cost-effective scaling of distributed servers, and as a result more directly supports the goals of the Commonwealth project.

Acknowledgments:

We would like to thank all members of the Commonwealth Research Group for their support and for the many useful discussions that helped solidify the results presented in this report. In particular, we would like to acknowledge the help of Mark Crovella, David Martin, Jun Liu, Jorge Londono, and Paul Barford. This work was partially supported by NSF research grant CCR-9706685 and by Microsoft.

References

1. A. Bestavros, M. Crovella, J. Liu, and D. Martin "Distributed Packet Rewriting and its Application to Scalable Web Server Architectures," in *Proceedings of ICNP'98: The 6th IEEE International Conference on Network Protocols*, (Austin, TX), October 1998.

2. A. Bestavros, M. Crovella, J. Liu, and D. Martin, "Distributed Packet Rewriting and its Application to Scalable Server Architectures," Tech. Rep. BUCS-TR-98-003, Boston University, Computer Science Department, February 1998.

3. K.L.E. Law, B. Nandy, and A. Chapman, “A Scalable and Distributed WWW Proxy System”, Nortel Limited Research Report, 1997.

4. Daniel M. Dias, William Kish, Rajat Mukherjee, and Renu Tewari, “A Scalable and Highly Available Web Server”, Proceedings of IEEE COMPCON'96.

5. Dahlin *et al*, “Eddie: A Robust and Scalable Internet Server”. Ericsson Telecom AB. Sweden. 1998.

6. Damani *et al*. “ONE-IP: Techniques for Hosting a Service on a Cluster of Machines”, Sixth International WWW Conference, April 1997.

7. Cisco Systems. “Scaling the Internet Web Servers”. A white paper available on the Web from Http://www.cisco.com/warp/public/751/lodir/scale_wp.htm. November 1997.

8. Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In Proceedings of ACM SIGMETRICS, 1998.

9. E.D. Katz, M. Butler, and R. McGrath, “A scalable HTTP server: The NCSA prototype. In Proceedings of the First International World-Wide Web Conference, May 1994.

10. D. Anderson, T. Yang, V. Holmedahl, and O.H. Ibarra. “SWEB: Towards a Scalable World Wide Server on Multicomputers”. In Proceedings of IPPS'96, April 1996.

11. Eric Anderson, David Patterson, and Eric Brewer. “The MagicRouter: An application of fast packet interposing.” Available from http://HTTP.CS.Berkeley.EDU/~eanders/projects/magicrouter/osdi96-mr-submission.ps, May 1996.

12. C. Perkins. “IETF RFC2003: IP Encapsulation within IP”. Available from Http://ds.internic.net/rfc/rfc2003.txt

13. Jeffrey Mogul, Richard Rashid, and Michael Accetta. “The Packet Filter: An Efficient Mechanism for User-level Network Code”. In Proceedings of SOSP'87: The 11th ACM Symposium on Operating Systems Principles, 1987.

14. Jeffery Mogul. “Network behavior of a busy Web server and its clients”. Research Report 95/5, DEC Western Research Laboratory, October 1995.

15. I^2O Special Interest Group. See Http://www.i2osig.com

16. IBM Corporation. “The IBM Interactive Network Dispatcher”. See Http://www.ics.raleigh.ibm.com/netdispatch

17. The Commonwealth Scalable Server Project. See Http://www.cs.bu.edu/groups/cwealth