RC 20853 (05/19/97)
Computer Science/Mathematics

IBM Research Report

NetDispatcher: A TCP Connection Router

G. Goldszmidt, G. Hunt

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

NetDispatcher: A TCP Connection Router

Germán Goldszmidt and Guerney Hunt
IBM T. J. Watson Research Center
Hawthorne, New York 10532
{gsg, gdhh}@watson.ibm.com

Abstract

NetDispatcher is a software router of TCP connections that supports load sharing across multiple TCP servers. It consists of the *Executor*, an operating system kernel extension that supports fast IP packet forwarding, and a user level *Manager* process that controls it. The Manager implements a novel dynamic load-sharing algorithm for allocation of TCP connections among servers according to their real-time load and responsiveness. This algorithm produces weights that are used by the Executor to quickly select a server for each new connection request. This allocation method was shown to be highly efficient in real tests, for large Internet sites serving millions of TCP connections per day. The Executor forwards client TCP packets to the servers without performing any TCP/IP header translations. Outgoing server-to-client packets are not handled by NetDispatcher and can follow a separate network route to the clients. Depending on the workload traffic, the performance benefit of this half-connection method can be significant. Prototypes of NetDispatcher were used to scale up several large and high-load Internet sites.

1 Introduction

As the global Internet and Intranet traffic increases, many sites are often unable to serve their TCP/IP workload, particularly during peak periods of activity. The main objective of *NetDispatcher* is to enable scalable TCP/IP server clusters that can handle millions of TCP connections per hour. This goal is achieved by routing incoming TCP connections to a set of servers, so that they share the workload efficiently.

NetDispatcher supports a collection of *Virtual* IP Addresses *(VIPAs)* that it shares with a set of hosts in a LAN. Each VIPA is shared by NetDispatcher and a *"Virtual Encapsulated Cluster" (VEC)*, a collection of hosts that provide the same function and serve equivalent content. The hosts of a VEC may have different hardware architecture and operating systems, as long as they support equivalent TCP/IP services. NetDispatcher assumes that each VIPA represents a VEC, but does not enforce this. It is the responsibility of the system administrators to ensure that equivalent function and content is provided by each of the VEC hosts.

For example, a simple VEC may consist of 10 hosts sharing the same VIPA and supporting only TCP port 80. Each of these hosts will execute Web server code accepting TCP connections on port 80 and delivering equivalent content. When a Web browser access a page, it may open several concurrent TCP connections, each of them may be routed to another VEC host. Hosts can be dynamically added or removed from each VEC via NetDispatcher configuration commands.

NetDispatcher uses a load-sharing algorithm that allocates connections in inverse proportion to the current load of each VEC server. This algorithm is implemented by two interacting components that work in tandem but at different rates. The *Executor* is a kernel-level extension to the TCP/IP stack, and the *Manager* is a user-level management tool.

Surges in Internet traffic tend to occur in waves, with intervals of heavy usage in which traffic peaks tend to cluster [Sta97]. The service time and the amount of server resources and network bandwidth consumed by each TCP connection request varies widely. To address these workload characteristics, NetDispatcher establishes a dynamic feedback control loop with the servers. The Manager dynamically monitors the current performance of the servers, evaluates a configurable estimate of each server's load, and computes in real-time the proportional allocations for each server. The Executor allocates new TCP connections proportionally to the Manager-computed weights, and then forwards following packets of each connection to the corresponding server.

We designed and implemented a NetDispatcher prototype running on RS6000

AIX 4.1 that supports heterogeneous networks and servers. This code was used to scale up several large Web sites, including the official Atlanta 1996 Olympic Web site. Laboratory tests and live measurements indicate that our initial prototype can handle over 200 million TCP connections per day (over 2300 connections per second). This prototype has been ported to other platforms and many performance optimizations have been implemented.

From several experiences using this tool to build large scale TCP/IP service sites, we found the following benefits. First, there are significant performance advantages in avoiding the header translations done by other tools. Second, dynamic load sharing enables efficient allocation of resources and reduces the service time of the requests. Third, configurable load metrics evaluated in real-time are necessary to provide customized feedback for each site.

The rest of this paper is organized as follows. Section 2 describes how NetDispatcher is used in TCP/IP networks. Section 3 describes the Executor's packet forwarding operation, the method to allocate new connections, and the internal instrumentation. Section 4 describes the design of the Manager, its dynamic feedback load sharing algorithm, the types of load metrics, and their configuration. Section 5 outlines our initial prototype implementation and its performance characteristics in supporting scalable Web sites. Section 6 describes some alternative approaches to solve the scalability problem, including client-based, 2 connections, DNS-based, packet forwarding, TCP header translations, and HTTP redirection. Section 7 discusses some current work and planned future enhancements, and Section 8 concludes.

2 Using NetDispatcher in a TCP/IP Network

This section outlines how NetDispatcher is used. The administrator of NetDispatcher must define the VIPAs, ports and servers of each VEC. All of this configuration information can be changed dynamically. The VIPAs become known via DNS and ARP, so that all IP packets for a VEC will be delivered to NetDispatcher. In contrast to DNS-based solutions, the private IP addresses of each server do not need to be propagated beyond NetDispatcher, where external hackers could use them for pernicious purposes.

2.1 Configuring the Servers

Each VIPA that a server supports must be aliased on an interface so that the local TCP stack will accept the corresponding incoming IP packets to that address. However, the servers may not export the VIPAs via ARP, since that would create LAN conflicts. Non-exported IP aliasing can be achieved in several ways. One method is aliasing the shared IP address to a non ARP-exported interface, like the *loopback* interface. These methods require the configuration of VEC server interfaces, (e.g. performing `ifconfig` commands). However, in contrast with other tools, these methods do not require the installation of any software nor O/S upgrades at the servers.

2.2 Network Configuration

NetDispatcher can be configured within one or several networks. Figure 1 shows a typical 2-network configuration of NetDispatcher. The *internal* network connects NetDispatcher (ND) to the servers (S1, S2, S3) and is used for NetDispatcher-to-server packets only. In this figure, the *external* network is used for both incoming client-to-NetDispatcher packets and for outgoing server-to-client packets. The client-to-NetDispatcher traffic could also be configured to arrive via a network interface that is connected directly to the IP router.

For instance, the internal network could be an Ethernet where only NetDispatcher introduces packets for the servers, the external network could be a Token Ring where servers forward their traffic, and the packets could arrive into NetDispatcher from an ATM link. In Figure 1 NetDispatcher has the VIPA 129.34.129.8 on the interface that connects it to the *"external"* network, and a private IP address 9.2.254.64 in the internal network. In S1, S2, and S3, the VIPA 129.34.129.8 has been aliased to their loopback interface.

2.3 Forwarding Packets

For each IP packet that represents a new TCP connection request, NetDispatcher chooses a server from the target VEC, and then forwards the packet to that server. Subsequent client-to-server IP packets for an existing connection are forwarded to the corresponding server. Outgoing server-to-client packets do not need to flow through NetDispatcher, but may follow a separate route, as illustrated by Figure 2. The target servers must be on a LAN segment that is directly connected to NetDispatcher, i.e., without any intervening IP router. Future support for forwarding packets accross IP

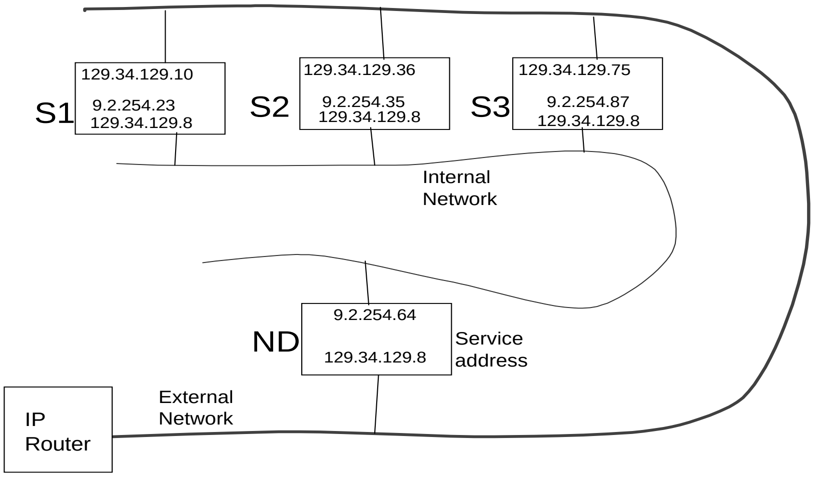

Figure 1: Sample network configuration for 2 networks.

routers is described in Section 7.

For example, the primary 1996 Atlanta Olympic Games Web site used 2 separate networks for incoming and outgoing traffic. A Token Ring was used for traffic from NetDispatcher to the servers, and ATM for traffic from the servers to 4 IP Routers that forwarded it via T3 links onto the Internet. Figure 3 illustrates the route taken by the IP packets of one TCP connection in this configuration. Bandwidth utilization can be more efficient by allowing the server-to-client IP packets to follow a separate route through different networks. This half-connection forwarding method is particularly useful for TCP-based protocols like HTTP, that are characterized by small client requests that may generate large server responses.

HTTP client packets are typically very short, e.g., requests to get a new page, and *acks* for the data received. Server-to-client packets are larger, as they include application content data. If the Web server delivers multimedia data, the ratio of outgoing to incoming bits is very large. NetDispatcher processes only the incoming packets. In contrast, header-translation tools (see Section 6.5) must process both incoming and outgoing packets. Hence, this is an obvious performance advantage for this type of traffic.

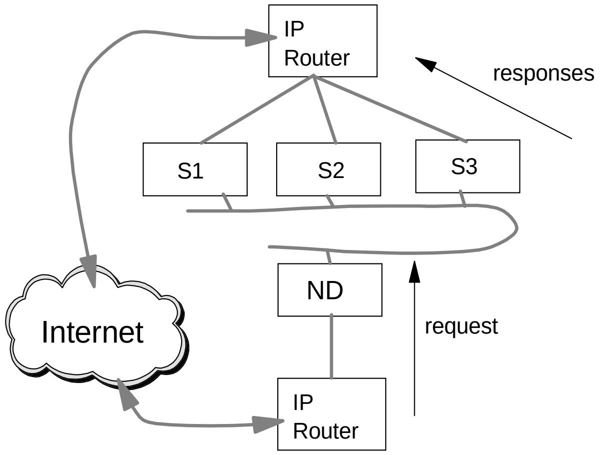

Figure 2: Example of NetDispatcher Traffic

2.4 Other Features

Filtering Packets. NetDispatcher acts as a TCP filter for the VIPAs by discarding many types of IP packets. For instance, it discards all IP packets destined for ports that have not been explicitly defined via configuration commands. For example, NetDispatcher can be configured to only allocate connections destined for TCP port 80, and discard all other packets.

Quiescing, and Co-location. NetDispatcher also supports quiescing servers. When a server is quiesced the currently existing TCP connections are not broken, but no new connections are assigned to the server. This feature is useful for dynamically upgrading the software and/or content on the servers. VEC server processes may also execute on the same host as NetDispatcher. This is very useful for low workload periods, during which NetDispatcher may be configured to allocate most connections locally.

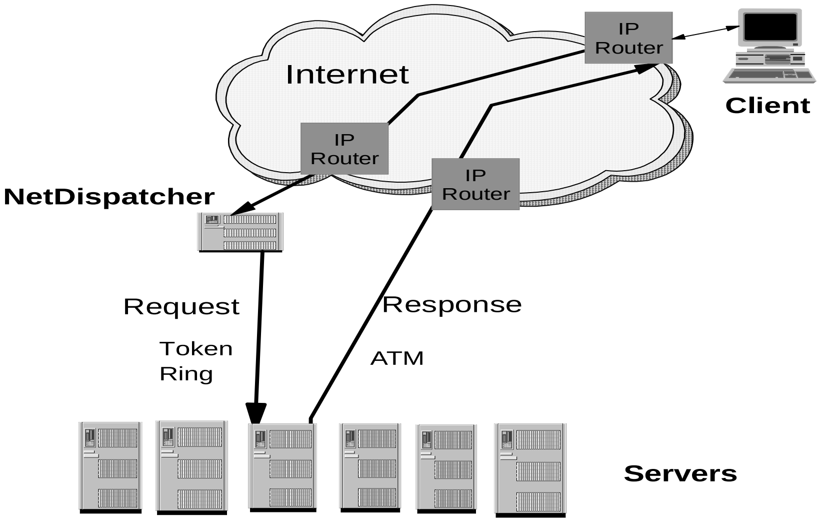

Figure 3: Overview of a NetDispatcher TCP connection route.

3 The Executor TCP/IP Extension

The *Executor* is a kernel-level extension to the TCP/IP stack that forwards TCP packets to VEC servers. The Executor code inserts itself between TCP and IP in the protocol stack, by taking over (and saving) the TCP input entry in the protocol stack switch table. Its main data structures are illustrated in Figure 4.

For each VEC, the Executor maintains a *port table* of all its active TCP ports. Each port P in a port table references a *server table.* Each entry in the server table represents a server S which is serving the port P for the VEC. Each server host entry is identified by a *private* IP address, and has a current weight value $W_P(S) \geq 0$, which is used to proportionally allocate new connections for port P as described below. Any server with $W_P(S) = 0$ is "quiesced" and will not receive any new connections.

A connection record is kept for each connection in a global hash table, the *connection table.* Each entry in this table is indexed by the source IP address and source port, and also includes the target server, a state indication (active or finished), and a time-stamp.

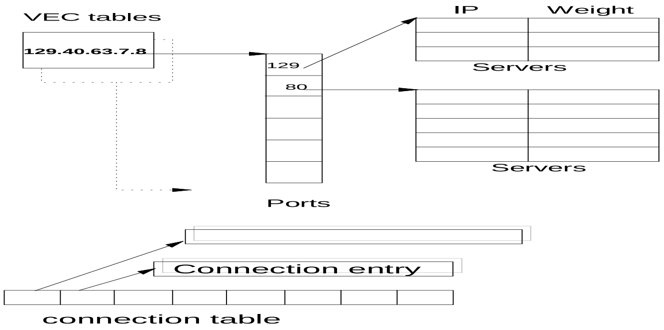

Figure 4: Main Data Structures of the Executor.

3.1 Packet Processing and Forwarding

All packets accepted by IP and destined for TCP are processed by the Executor. Each packet can be either (1) discarded, (2) handled by the local TCP stack, or (3) forwarded to a server for a new or existing connection. If the target port is not defined for the VEC, or if there are no servers currently configured for that port, the packet is discarded. If the packet's target IP address is not the same as one of the VECs, the packet is passed to the local TCP stack for processing.

3.1.1 Existing Connections

For each incoming packet, a hash function is computed on the source (client) IP address and source port to find an existing TCP connection in the connection table. Because of this, the hash computation must be very efficient, and should result in a low collision rate. The inlined hash function is computed using only shifts, ANDs, and ORs instructions. The input of the hash function is a subset of the bits of the source port and source IP address. At a popular Internet site that served some 40 million TCP connections per day, we observed a collision rate of 2% with a hash table of 16K entries.

If the TCP packet is for an existing connection, the Executor checks the flags of the TCP header. If the RST flag is set, the connection entry is purged from the connection table. If the SYN (FIN) flag is set, the state of the connection is changed to active (finished). Then the packet is forwarded to the corresponding server. If no connection is found and the packet does not have the SYN flag set (new connection request), it is discarded. If a TCP packet has both the SYN and FIN flags set, the connection is created and then marked finished.

3.1.2 New Connections

A Weighted Round-Robin (WRR) algorithm (see Section 3.2) is used to select a target server for a new TCP connection request. For each port P, served by a sequence of n servers $\{S_n\}$, WRR uses a sequence of n weights $W_P(S_n)$, that are concurrently computed by the Manager (see Section 4). WRR distributes connections based on a ratio of the weights. For instance, assume that $W_P(S_A) = w$ and $W_P(S_B) = 2w$. Then, while the ratio $W_P(S_B)/W_P(S_A)$ is maintained, S_B will get twice as many new connections to port P as S_A. After the connection table entry is made, the packet is forwarded to the indicated server.

3.1.3 Packet Forwarding

The Executor uses the *private* IP address of the selected server to find the network interface on which each IP packet should be forwarded. If no interface is found for that IP address, the packet is discarded. Otherwise, the Executor constructs a network frame for transmission to the indicated interface, by invoking existing O/S services, without examining the contents of the packet. If the IP packet is too large to fit in one frame of the target network, the packet is fragmented. The forwarded frame has the MAC address of the network interface of the selected server.

3.1.4 Port Specific Handling

There are some TCP-based protocols that require special attention, since they implicitly create *sessions* that consist of multiple TCP connections between a single client and a single server. For example, FTP spans two ports, the control and data port, and both connections should be routed to the same server. The Executor allows for specific handling of these cases, which are identified by their port numbers. For FTP control connections, the Executor does the connection lookup based only on the client's IP address, so that all the connections from a given client to the same VIPA are routed to the same FTP server. When a connection request arrives for the FTP data port, the Executor first checks to see if a control connection exists from the same client to any server. If it does, the new FTP data connection is routed to the same server as the control connection, otherwise it is handled like any other TCP connection. There are many other protocols that require client affinity to the same server, e.g. SSL.

3.2 Weighted Round-Robin Allocation of New Connections

The Weighted Round-Robin (WRR) is a simple and efficient algorithm for selecting a new server. For each active port P in a VEC, WRR uses the following data structures: (1) an ordered circular list of host servers $L = (S_1, S_2, ..., S_n)$, (2) an index i to the last selected server in L, (3) a *maximum* weight variable ($mw > 0$), and (4) the *current* weight ($0 \leq$cw$\leq mw$), first initialized to mw. If $\forall i,\ W_P(S_i) = 0$, there are no available servers, and all incoming TCP connection requests for port P are discarded.

The main WRR loop starts by incrementing i. Whenever $i = 1$ (the "first" server in L), $cw \leftarrow cw - 1$. If $cw = 0$, $cw \leftarrow mw$. If $W_P(S_i) \geq cw$, return S_i as the selected server to allocate the next connection, (and cw remains unchanged for the next invocation of WRR). Otherwise, ($W_P(S_i) < cw$), $i \leftarrow i + 1\ mod\ |L|$, and WRR loops. This loop will always terminate, because $mw > 0$, and at least one server S has $W_P(S) = mw$.

If the weight changes are infrequent, and/or $|L|$ is large, the above algorithm can be improved as follows. First, the servers in L should be maintained in decreasing order by weight ($mw = W_P(S_1) \leq W_P(S_2) \leq ... \leq W_P(S_n)$). Second, when the weights are changed by the Manager, the algorithm is reset by $i \leftarrow 0$ and $cw \leftarrow mw$. Third, if $W_P(S_i) < cw$, then $i \leftarrow 0$ and $cw \leftarrow cw - 1$.

3.3 Connection Termination

Because the Executor sees only the client-to-server half of each TCP connection, connection termination cannot be completely determined. Hence, both client initiated "active" closes and client failures can result in accumulation of stale entries in the connection table. Let us examine each of this two cases:

- When the client closes the conversation before the server, ("active close") the connection should not be closed immediately. A client's active close implies a request that the server close its half of the conversation after sending all the requested data. ACK packets from the client to the server must continue to flow through NetDispatcher. Therefore, the connection record should not be purged when NetDispatcher first observes a client's FIN.

- If a client looses connectivity to NetDispatcher (e.g. crashes) there may be no FIN or RST on the connection. The server will eventually time-out this connection, but the connection table will still have this stale connection entry in active state.

Both active-close and stale connections have configurable time-out periods. These timeouts specify the minimum idle time period since the current connection timestamp, before a connection in finished or active state can be purged. Both client initiated "active" closes and client failures can result in accumulation of stale connection table entries. The Executor handles the problem of accumulation of stale connection table entries by time stamping the connection records each time a packet flows through them, and by garbage collecting them when they have been idle for a configurable amount of time. The garbage collector thread is activated in the background by the TCP slow timer.

3.4 Instrumentation Counters and Gauges

The Executor maintains two gauges and two counters for each server. These variables can be queried by the Manager (see Section 4.3) to estimate the current load on the servers. The gauges keep the current number of connections that are in active and finished state. The counters keep count of the total number of connections a server has received and the total number of connections that have been completed.

For diagnostic purposes the Executor also keeps packet counters, such as the number of packets that were received, too short, discarded, forwarded, and erroneous. Also, for each VEC similar counters are kept, including the number of SYN packets received on active (finished) connections, and the number of packets dropped. These counters were used to monitor the effects of the different time-out values on the operation of the Executor. These counters could be used by a private MIB (Management Information Base) to support SNMP management of NetDispatcher.

4 Managing Load in NetDispatcher

The Manager component of NetDispatcher defines the policy of dynamic load-sharing connections among the VEC servers. The load sharing policy takes into account the real-time load and responsiveness of the servers. Figure 5 illustrates the relationships between the Manager, Executor, Advisors, and servers. *Advisor* processes collect and

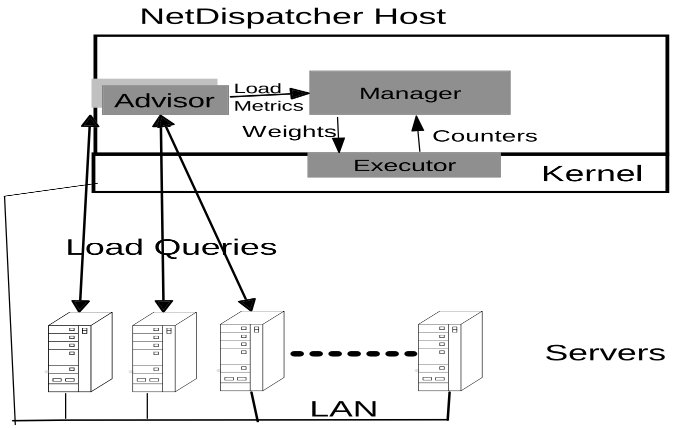

Figure 5: The NetDispatcher Manager

order load information from the VEC servers. The Manager uses the load measurements provided by the Executor and the Advisors to compute a configurable *Load Metric Index* (LMI) value for each service port. Using present and past LMI values and the current weights, the Manager computes a new set of weights. If these weights differ by more than a threshold from the current weights, they are assigned to the Executor's data structures to be used by the WRR algorithm.

This Section first motivates the need for real-time dynamic feedback load sharing by describing how the imbalance in request processing time can often cause skewed utilization of the cluster resources. We then outline the differences between load balancing and load-sharing, describe the types of load metrics used, and how the LMIs and weights are computed, and discuss the effectiveness of the load-sharing method.

4.1 Dispatching TCP Connections

TCP connections are used by remote clients (e.g., Web browsers) to request services from VEC servers (e.g. Web Servers). The service time and the amount of resources consumed by each request varies widely and depends on several factors, such as the type of service being provided, the specific content involved in each request, the current network bandwidth available, and others. For instance, a complex database query will probably take orders of magnitude more time than providing a static, text-only, pre-loaded HTML page. Some "heavy" requests perform long transactions that involves computational intensive searches, database access, and/or very long response data streams. Lighter requests may involve fetching an HTML page from cache or performing a trivial computation.

This imbalance in request processing time often causes skewed utilization of the server cluster. For instance, consider an application that opens 4 concurrent connections to retrieve 4 graphic files, A, B, C, and D, one of which (D) is very large. Assume that several independent clients execute the same code. It is possible that some of the cluster servers will be running at capacity while others are mostly idle. As an extreme example, all the D file requests may arrive at the same server. If one or more connections are queued on a busy server, the client application may time out.

Simple Allocation of TCP Connections. A naïve distribution of TCP connections among the back-end servers can (and often does) produce a skewed allocation of the cluster resources. In the previous example, a simple round-robin allocation scheme may result in many requests being queued up on servers that are currently serving heavy requests. For instance, if there were 4 servers in the cluster, one of them would always receive the request for the large file (D). Such an allocation policy can cause a severe underutilization of the cluster resources, as some servers may stay relatively idle while others are overloaded. This condition, in turn, will also produce longer observed delays for the remote clients.

Actual TCP/IP traffic patterns. Surges in network traffic tend to occur in waves, with long periods of little traffic followed by intervals of heavy usage in which traffic peaks tend to cluster [Sta97]. This self-similarity of network traffic has been shown to apply to both WANs and LANs in general, and to particular subsets, e.g., Web traffic [CB95]. Some form of dynamic feedback control is therefore necessary to enable appropriate reaction to the actual traffic patterns and the state of the servers. This feedback could be used by the clients, the servers, or an intermediary like NetDispatcher to more evenly utilize the cluster resources. Implementing this in NetDispatcher has the advantage of being transparent to both clients and servers.

4.2 Load Sharing with Dynamic Feedback

Load Balancing and Load Sharing. NetDispatcher needs some guidance in order to allocate the TCP-connections in a way that utilizes the cluster resources efficiently. There are two techniques for improving performance using load distribution among several servers: *load-balancing* and *load-sharing* [ELZ86]. Load-balancing strives to equalize the servers workload, while load-sharing attempts to smooth out transient peak overload periods on some nodes [KK92]. Load-balancing strategies typically consume many more resources than load-sharing, and the cost of these resources often outweigh their potential benefits [KL87].

The TCP/IP workload that we observed at several heavy load Internet sites was characterized by having many very short transactions and a few long ones. This was the case for the Olympic Games Web site. For this type of workload, NetDispatcher used a load-sharing method that proved to be very efficient, and also achieved a relatively uniform distribution of the workload. If the workload characteristics were to change significantly, so that longer transactions dominate, load balancing should be applied to achieve a more uniform distribution of the workload.

Load Sharing. NetDispatcher implements a load-sharing allocation policy for new TCP connections, which is driven by a dynamic feedback control loop with the VEC servers. The Manager monitors and evaluates the current load on each server using combinations of load metrics, as described in Section 4.3 below. The Manager uses this data to compute the weights associated with each port server instance in the Executor's WRR algorithm. The computed weights tend to proportionally increase (decrease) the allocation of new TCP connections to underutilized (overutilized) servers.

By controlling the assignment of these weights, the Manager can implement different allocation policies for spreading the incoming TCP connections. For instance, an allocation policy may assign $\forall i \neq j,\ W_p(S_i) = 0\ , W_p(S_j) = 1$. In this case, all the traffic for port p will be sent to host j. By choosing j to be the least-loaded host, the Manager can implement a *best* server allocation policy. Such a policy may be very efficient during periods of relatively low load. Also, an administrator may decide to quiesce a server ($W_p(S_i) = 0$) while doing some maintenance on it, or when discovering that its application state is corrupted.

4.3 Load Metric Index

The Manager computes a configurable *Load Metric Index* (LMI) value for each service port. This computation estimates the current state of each server using multiple load metrics and administrator configurable parameters such as time thresholds. Load metrics are classified into three classes, according to the way in which they are computed: *input, host,* and *forward.* Input metrics are computed locally at the NetDispatcher host. Host metrics are computed at each server host, and forward metrics are computed via network interactions between NetDispatcher and each server host. These metrics could be further combined to provide a *health index* [GY93] for network management purposes.

4.3.1 Input Metrics

Input metrics provide a current estimate of the state of the VEC servers, as seen from NetDispatcher. These metrics are derived from the counters and gauges that the Executor collects (as described in Section 3.4). For example, the number of new connections received in the last t seconds is an input metric. To compute this metric the Manager periodically retrieves the values of the corresponding Executor counters. By subtracting two counters of a server polled at times T_1 and T_2, the Manager computes a metric variable that represents the number of connections received during the interval $T_1 - T_2$. The aggregation of such input metrics provides an approximation to the current rate of new connection requests for each server and each port service.

4.3.2 Host Metrics

Host metrics represent the load of a server's host. The total number of active processes, or the total number of allocated mbufs in a given server are examples of host metrics. Host metrics are typically computed at each host server by an *agent* process that executes command scripts. These scripts return numerical values which are reported to a corresponding *Advisor* process at the NetDispatcher host. The Advisor collects and orders the reports from all the hosts and periodically presents them to the Manager.

If a metric report is not received within a policy-specific threshold time, then the corresponding host metric is given a special "unreacheable" value. The Manager may then decide to temporarily quiesce that host, h, by assigning $W_p(S_h) \leftarrow 0$ for all its active ports, so that no new connections are forwarded to it.

For the Olympic Games Web site, NetDispatcher used a configurable Advisor to collect host metrics. For example, the "number of active processes" metric was computed by the command " `ps -ef | wc -l`." This string, together with a time value representing the period between computations, and a list of hosts and other configuration values could be given to an Advisor process. The Advisor sent the command string and its configuration parameters to all the corresponding host agents.

The main advantage of this method is the ability to tailor the script to measure very specific load metrics. The Olympic Games Web site used several different scripts for different workloads. For example, for a given memory intensive workload we measured the utilization of memory buffers (mbufs) for network connections.

The main disadvantage of the scripting method is its potentially high overhead, particularly when the metrics must be updated very frequently and when the hosts have very high CPU utilization. This overhead includes the computational cycles spent in the frequent execution of the scripts at the servers, and the network bandwidth spent on exchanging metric reports.

4.3.3 Forward Metrics

Forward metrics are computed by sending messages from NetDispatcher to a specific host service. For instance, the time required to retrieve an HTML page from a Web server to the NetDispatcher host is a forward metric. An HTTP Advisor at the NetDispatcher host can send an HTTP "GET /" request to each Web server in a VEC, and measure their corresponding delays. Such a metric measures an approximation of the retrieval time that includes all the relevant factors: the actual instruction path through the service application, the time that the request spends in the different queues at the server host, LAN access time, and so forth.

If a request is not answered by a configurable time-out, the corresponding service is marked as temporarily not receiving new requests of the particular service type. The Manager can then decide that a service at a particular host is temporarily unreachable, and hence no more new connections of this type should be forwarded to it.

4.4 LMI Configuration and Computation

Host metrics typically take longer to acquire than forward metrics, which take longer than input metrics. The relative importance of each load metric can depend on the workload and services, which can change over time. Hence, the Manager enables dynamic configuration of relative metric weights $R(i)$, $(\sum R(i) = 1)$, to be associated with each load metric $l(i)$. $R(i)$ defines the relative importance of each $l(i)$ metric for the Manager's load allocation algorithm. The combination of all the weighted metric instances is an aggregate load metric index $L_P(S) = \sum R(i) * l(i)$, for each server S and port P.

For instance, a network administrator may configure the algorithm for port 80 to give 20% relative weight to a given host metric A, 40% to a service metric B, and 30% and 10% to two distinct input metrics C and D. Hence, for each server S, its computed LMI for port 80 is $L_{80}(S) = 0.2 * A + 0.4 * B + 0.3 * C + 0.1 * D$.

The $R(i)$ configuration weights can be changed at any time. Ideally this should be done by an automated management tool. The "best" allocation depends on installation specific parameters which are dynamically tuned to the changing nature of the system workload. Network administrators can dynamically adjust many of the parameters of the algorithm to compute the LMI. For example, administrators may dynamically change the $R(i)$ weights to raise the influence of a host metric (e.g. buffer utilization) while lowering the influence of a forward metric (e.g. HTTP page retrieval). We found this feature to be extremely useful during the Olympic games. As the workload and the content of the servers changed, we were able to compose a more useful LMI.

4.5 Weights Computation

The weight assignments are computed at a configurable periodic interval (e.g. 5 seconds), and whenever a significant event occurred (e.g., when a new server was added). The first step is the normalization of all the LMIs and the current weights. For each active server instance, it takes the previous and present LMIs, and the current weights, and computes their proportion of the total.

The second step actually computes a new vector of weights using a replaceable *Weight Computation Function (WCF)*. The WCF takes as input all the above proportions, and some additional parameters (e.g. a smoothing factor used to prevent strong oscillations). The third step is to compute an (absolute) aggregate of all the weight changes for each port service. If the aggregate is more than a configurable "sensitivity" threshold, the new assignments are committed, that is, they are assigned to the Executor's tables. If the weight changes are less than the threshold, we avoid the overhead of interrupting the Executor.

4.6 Discussion

Because the Manager implements a load sharing algorithm, at any particular point in time there may be some imbalance in the utilization state of the servers. This is unavoidable for many types of Internet service workloads (e.g. Web pages), as these "transactions" typically complete very quickly, before the relevant feedback can reach NetDispatcher. In other words, the delay in the feedback loop is too long compared with the average transaction size to enable effective load balancing.

The load sharing algorithm was sufficiently sensitive to host overload at the Olympic Web site. The Manager provided a very effective combination of the different types of metrics. Its algorithm converged rapidly, did not create oscillations, and resulted in overall lower average wait time for requests. Whenever a host h became overloaded, the input metrics indicated an abnormal number of connections. This, in turn, resulted in a high number for the combined load metric for that host. When this occurred the host was quickly quiesced, i.e., its weight was automatically set to 0. During this "cool-down" period, most hosts were able to resolve whatever temporary resource allocation problem they had. By then the host metrics indicated that the host was underutilized, lowering the combined load metric. At this time the Manager automatically assigned a positive weight to h, reintroducing it to the VEC working set.

Notice that any host that becomes overloaded can have an impact on the externally perceived quality of service of the site, as requests to that host may stay queued up for long periods of time. Quiescing the hosts in the above fashion prevents incoming request from hitting temporary unresponsive servers (e.g. "Connection Refused").

5 Implementation and Performance

5.1 Background

An early version of the IP packet forwarding technology in the Executor was developed as part of the *Encapsulated Cluster* (EC) project [AS92]. In the EC project, incoming IP packets were forwarded to the cluster servers by rewriting the packet headers. The corresponding outgoing packets had their headers rewritten so that they appeared to come from a single IP address. They also described [AS94] and partially prototyped a version that supported half-connection traffic. A prototype implementation kernel extension TCP connection router, called the (*"TCP-Router"*), was based on a subset of the EC code. This prototype supported only one shared IP address, and used a simple round-robin method to allocate TCP connections. This code was integrated into a prototype parallel Web server that required a separate internal network. A simulation analysis of the expected performance of that Web server is given in [DKMT96].

5.2 NetDispatcher

We started the NetDispatcher project by defining the basic requirements to support large scale TCP/IP service sites. Among them, support for multiple VIPAs and ports, dynamic real-time load management based on the actual state of the servers, fast allocation of connections and IP packet forwarding, and extensive dynamic configuration support. Parts of the Executor re-used some C source code from the original TCP/IP stack, the EC project, and the TCP-Router. This involved restructuring the software, fixing bugs, and performance optimizations of the TCP-Router. NetDispatcher added significant functionality in this kernel extension, a new control interface, dynamic real-time load management, and extensive dynamic configuration support. Among the enhancements that we introduced to the kernel extensions are:

- complete support of half-connection traffic via NetDispatcher,
- the WRR algorithm for real-time weight-based allocation of connections,
- efficient hashing functions,
- garbage collection,
- monitoring counters,
- support for multiple VIPAs, each with its own set of ports, and each port with its own set of servers, and
- support for client affinity on a specific port.

5.3 Implementations

A prototype of NetDispatcher running on a RS6000 AIX 4.1 was used to implement several large scale Web sites since early in 1996. This prototype has been used with Ethernet, Token Ring, FDDI, and ATM networks, supporting heterogeneous servers, including OS/2, several Unix platforms, and Windows NT. An early prototype of

NetDispatcher was available in late 1996 for Internet downloading from the AlphaWorks site, in `www.alphaworks.ibm.com`. Since then, the NetDispatcher code has become part of IBM's *Interactive Network Dispatcher* product [1], and has been ported to other platforms, including Windows NT, Solaris, and two hardware IP routers.

5.4 Prototype Performance

Olympic Web Site. In the official Atlanta 1996 Summer Olympic Web site, one NetDispatcher host supported 5 VECs on some 50+ SP/2 nodes. This Web site was geographically distributed among 5 locations in the USA, Europe and Asia, each location containing one NetDispatcher host. During the games, this prototype handled over 190 million TCP connections (HTTP hits) at the main location in Southbury.

At this site it executed on an RS6000 model 39H (67Mhz POWER2 processor). It was connected to two 16MB Token Rings, one for incoming Internet traffic from the IP routers, and the other for spreading the packets to the servers. Response packets from the servers were sent via ATM links to IP Routers that forwarded them via four T3 links onto the Internet. This topology took advantage of the half-connection method of NetDispatcher to better utilize the available network bandwidth.

During peak activity periods, NetDispatcher handled over 500 HTTP TCP connections per second with low CPU utilization (under 5%) The additional latency overhead involved for each connection was negligible, particularly when compared with the latency of typical HTTP Internet connections.

Netscape. At the Netscape Web site during the summer of 1996, the prototype handled an average of 40 million TCP connections per day on an 8 node SP/2 using Ethernet and an SP/2 switch. In this popular Internet site collision rates of approximately 2% for a hash table of 16K entries were observed.

Laboratory tests. A series of laboratory measurements were conducted using Webstone [TS95] and running NetDispatcher on a similar processor and using an SP/2 switch [MM96]. In this study, NetDispatcher supported over 8 million HTTP requests (TCP connections) per hour (almost 200 million per day) for small file requests. These measurements indicate that one NetDispatcher host can support hundreds of TCP servers.

[1] See http://www.ics.raleigh.ibm.com/netdispatch/ .

Optimizations. Since the above measurements were taken, significant optimizations on the Executor have been implemented. Initial measurements of this improved code suggest a significant speed-up on the kernel's packet processing, resulting in only 20% of the CPU cycles previously used on packet processing. Hence we expect higher throughput and also lower latency in the latest implementation.

6 Alternative Approaches

Alternative solutions to the scaling problem can be classified into the following main categories:

1. Client-based choice of server by the end-user or implicitly by the software.
2. Splitting each TCP connection into two TCP connections.
3. DNS-based, which enable clients to implicitly choose a server;
4. Simple forwarding of IP packets, source-based or by LAN broadcasting.
5. Packet forwarding with TCP/IP header translation.
6. HTTP specific methods like client redirection or TCP connection hopping.

In the following we outline the main characteristics of these methods, and some of the tools that implement them.

6.1 Client Based Methods

Client-based schemes require the client software or the user to choose an appropriate server. They can be divided into 3 main groups, explicit by the end-user, implicit using pre-defined addresses, and specialized clients.

6.1.1 Explicit by End-User

For example, some Web pages present a list of servers and ask the user to choose one by clicking on the corresponding link, e.g.,
Please select one of the following: `s1.c.com`, `s2.c.com`,...., `sn.c.com`.
Of course, there is no way to predict that end-users will produce a fair distribution of requests accross the servers. In practice, it is likely that this distribution will be very skewed towards a few servers (e.g. the first and last on the list).

6.1.2 Implicit with Pre-defined Addresses

Some client software (e.g. Netscape's Web browser) automatically chooses a random server from a given collection. This requires built-in knowledge, in the client, of the set of IP addresses of the available servers. Obviously, this method can only work for a limited set of sites and servers. Neither the explicit nor the implicit method takes into consideration the current load or availability of the servers. Hence, they are not sufficient solutions to the scaling problem.

6.1.3 Specialized Clients

The *Smart Client* project [YCE$^+$97] implemented specialized client software (telnet, ftp, and chat) that can select the servers. Applets provide service-specific customizations that choose a server for each new connection. These applets collect data about server load at several sites by making explicit requests. Remote measuring provides a delayed approximation of the current state of the servers. A potential drawback of this scheme is the amount of Internet traffic required to measure the state of the servers.

6.2 Two TCP Connections

Another method of distributing load among a set of servers is to create a second connection for each TCP connection. For example, Songerwala and Levy [SL96] prototyped a broker tool that accepts each TCP connection and allocates a second TCP connection between itself and a Web server host. Walker's *pWEB* [Wal96] follows a similar approach using a forwarding agent that accepts client TCP connections, appropriately labels the incoming messages, forwards them to servers, receives the responses from the servers and forwards them back to the appropriate client.

Two connection schemes have been used in the past for other purposes, such as to bridge wired and wireless networks and to implement proxy servers. The main advantage of the two-connection approach is its ease of implementation and portability. Its main disadvantage is its performance, in both latency overhead and lower throughput. Furthermore, such a scheme may violate the intended semantics of a TCP connection, since a client may get acknowledgments for packets that have not been actually received by the server [Tan96].

6.3 DNS extensions for Load Balancing

Many Internet sites rely on DNS [Moc87] based techniques to share load across several servers. A primary example is the NCSA Web site [KBM94]. These techniques include modifications to the DNS BIND code, like Beecher's Shuffle Addresses, Rose's Round-Robin code (RR-DNS), Brisco's zone transfer changes [Bri95], and Schemers' LBnamed [Sch95]. These techniques allow the DNS client to choose the IP address of the target server.

There are several drawbacks to all DNS-based solutions. Mogul, observing a high-traffic 3 server Web site, found that DNS-based schemes do not balance the loads over short periods of time. He suggested that "DNS-based technique cannot provide linear scaling for server performance at peak request rates" [Mog95].

DNS may only disclose up to 32 IP addresses for each name, due to UDP packet size constraints. Revealing these IP addresses may be problematic for some service sites, since this information can be used by hackers and competitors. It also makes it more difficult to change the IP addresses of the servers when needed. Knowledge of so many IP addresses also creates problems for clients (e.g., unnecessary reload of Web pages in cache), and for network gateways (e.g. for filtering rules in IP routers).

Any caching of IP addresses resolved via DNS creates skews in the distribution of requests. For instance, consider the case of a large ISP proxy caching the IP address of a server S for the DNS name D. All requests for D from that ISP will then be forwarded to S. In general, whenever server IP addresses become known, the cluster loses control on how many requests will reach each server.

While DNS allows servers to request that names expire immediately by setting a very short or even 0 TTL value, this option is oftentimes ignored. If secondary DNS servers cannot cache data, it results in (1) heavy traffic of DNS queries between the DNS servers, and (2) the primary DNS server becoming a single point of failure [Chu96]. Furthermore, there is a potential for high delay in resolving these names between the servers. Another problem of the DNS-based solutions is that they are very slow (or unable) to detect server failures and additions of new servers. In these cases, TCP clients will be refused connections while the cluster could have served them. Finally, DNS records can be spoofed to mislead a client into going to a phony server.

The simpler RR-DNS treats all servers equally, without regard to their power or current capacity. Hence, in an heterogeneous environment, an underutilized Pentium Pro server host may get as many requests as an overloaded 486 server. While the server host may be running properly, the specific service software (e.g. httpd) may have failed. For example, a tool like LBnamed may see that a server is underloaded (because the httpd demon failed), and then it will give it an even higher priority, resulting in more requests going to a failed server. Furthermore, there may be a large delay between the time of the actual measurements of load at the servers and the time at which the DNS requests are resolved. By then, the load information may be obsolete.

6.4 Simple IP Packet Forwarding

6.4.1 Source-based Forwarding

ONE-IP [DCH$^+$97] supports IP packet forwarding based only on a hash function over the source IP address, e.g. client-based affinity. Notice that there is no guarantee that the source IP addresses of the clients will be evenly distributed. In particular, many clients may reuse an IP address associated with proxy servers or other filtering gateways. Such a scheme will have some of the same problems of skewing that were described earlier for DNS based methods, and hence will not properly balance the load. Its main advantage is that it does not need to keep track of the connections, hence it requires less memory.

6.4.2 IP Packet Broadcasting

This method will forward all IP packets arriving for a VIPA to the broadcasting address of the LAN. Filtering software at each server then decides whether or not to accept the packets. The filtering code must properly divide the IP address space so that there are no overlapping across all the interfaces. This method was proposed by [Chu96] and was also prototyped in ONE-IP [DCH$^+$97].

6.5 TCP/IP Header Translation

Most packet forwarding tools are implemented as specialized hardware devices, e.g., Cisco's LocalDirector [CIS96]. These devices translate (rewrite) the TCP/IP headers and recompute the checksums of all packets flowing between clients and servers in both directions. This is similar to Network Address Translation (NAT) [EF94], adding the choice of the selected server. HydraWeb [Hyd96] is another similar device, but also requires installing software agents at each server to analyze their load.

There are several performance disadvantages of the header-translation approach. First, there is the latency overhead involved in processing all packets in both directions. Second, there are the bandwidth constraints of the translator device which becomes a main traffic bottleneck. Third, since each return packet must return via the translator device, it is not practical to have multiple devices working concurrently with a common set of host servers.

In contrast to this approach, NetDispatcher *half-connection* method leaves the incoming IP packets intact, and does not process the outgoing packets. The half-connection method has some significant performance advantages over the above header translation. The latency overhead is smaller since the IP packet is left untouched, and outgoing packets can follow several routes, which may have higher aggregate bandwidth. Since each return packet does not need to return via the forwarding NetDispatcher, it is very easy to have multiple NetDispatcher devices concurrently serving traffic to the same set of nodes. For example, 2 NetDispatcher devices can provide high availability for each other, and also work concurrently supporting the same set of host servers. Furthermore, the half connection method allows for passing the packets to remote LANs, as described in Section 7.

Several other research projects have also reported similar features. The Magicrouter [APB96] maps kernel memory to user space and allows direct access to packets of Linux device drivers. It translates the TCP headers (addresses and checksums) to forward the packets in both directions. It allocates connections using either round-robin, random, or incremental system load methods. Yeom et al. [YHK96] proposed a port-address translator gateway that assigns a different port number to each connection. Its translation of IP addresses and ports brakes many protocols which require affinity between several connections.

6.6 HTTP Specific Methods

6.6.1 Connection Passing for HTTP

Resonate Dispatch [Res96] is a software tool that supports content-based allocation of HTTP requests. TCP connections are accepted at the "Scheduler" host which examines the URL and then transfers the TCP connection to a host based on administrative rules. This method, however, requires modifying the IP stack of all the (SPARC/Solaris) servers to support the TCP connection "hop".

6.6.2 Selective HTTP Redirection

In this scheme, when a Web browser GET request reaches a specialized server, instead of returning the requested data, the client is directed to create a second connection to another server. Notice that a simple, "always redirect" policy can have a very high latency overhead for most simple requests. Hence, HTTP clients should be redirected to other servers only when the load in the original server is too high and the expected redirection overhead is smaller than the expected gain in performance. Obviously, this method is only useful for HTTP, and cannot address other legacy TCP/IP protocols.

SWEB [AYHI96] uses connection redirection in conjunction with RR-DNS to avoid hot-spots. Genuity's *Hopscotch* [JH97] also uses HTTP redirect to point the client to a different server over their internal network. It uses `traceroute` to build a large central database of network traffic, in order to find an appropriate server for each client request. Redirection may be very slow when a request comes from a previously unknown domain, as it will take very long to choose a server.

7 Current and Future Work

There are several additional features that may be integrated in future versions of these technology. Some of these features have been prototyped:

1. *High availability (HA)* enables clients to re-establish new connections to the servers quickly after a failure of NetDispatcher; In the event of a failure, however, HA will lose the currently active connections. HA scripts based on Phoenix technology [IBM96b, IBM96a] were used for an IBM SP/2 cluster.

2. *Fault tolerance* supports complete recovery without (or with minimal) loss of active connections. This scheme will enable recovery from a NetDispatcher failure without losing connections.

3. Additional port-specific routing functions, to support different affinity requirements; SSL support has been implemented.

Other features have been designed but not fully implemented yet. These include:

1. Support for wide-area network routing of TCP connections. This is done by tunneling incoming IP packets to a remote NetDispatcher that delivers them to hosts in its LAN. Return packets will go directly to the client.

2. Support to identify and properly respond to SYN flood denial of service attacks. An installation may have some servers which can properly handle such an attack (set A), while some other servers (set B) can not. Hence, an administrator may produce a policy that requires that set B hosts should not receive any TCP connections whenever a SYN flood attack is suspected.

8 Conclusions

Many popular Internet and Intranet sites need to scale-up to serve their ever increasing TCP/IP workload. Client-based and DNS-based methods, by themselves, have been shown to be insufficient ways to scale-up heavily used server sites, particularly during peak periods of activity. In general, whenever server IP addresses become known, the site loses control of the allocation of requests to each server. Load allocation decisions should be made in real time, as each request arrives, in close proximity to the cluster, and based on the current state of the servers.

NetDispatcher supports the sharing of a virtual IP address by several servers, and properly distributes the workload among them. Its method of forwarding incoming client-to-server TCP packets unchanged is more efficient and scales-up better than alternative TCP header-translator tools. For many TCP-based protocols (like HTTP), incoming request packets are typically smaller than the response packets. Outgoing server-to-client traffic can follow a separate network route, and need not be processed by NetDispatcher. Hence, its half-connection method provides a performance advantage over address translation in terms of bandwidth utilization and latency. Depending on the workload traffic, the performance benefit can be significant.

NetDispatcher supports dynamic, real-time feedback to properly react to the current traffic pattern. Instead of trying to find a server for every connection as the requests arrive, NetDispatcher uses an efficient user-level and kernel-extension combination load-sharing algorithm based on a configurable Load Metric Index value for each service port. This load-sharing scheme was shown to be efficient in real life tests, for large Internet sites with millions of TCP connections per day, such as the Summer Olympic Web site. In laboratory tests, an early prototype supported over 200 million TCP connections per day.

Acknowledgments

The basic technology for forwarding IP packets implemented in NetDispatcher was developed as part of the Encapsulated Cluster (EC) project by Clement Attanasio and Stephen Smith. Bill Kish started the development of a prototype *"TCP-router"* based on a subset of the EC code. The design and implementation of the NetDispatcher prototype *"WOMSprayer"* that was used at the Atlanta 1996 Olympic games Web site was done by Germán Goldszmidt and Guerney Hunt, with some assistance by Richard King and Ashutosh Tripathi. The Host Advisor was designed and implemented by Andy Stanford-Clark and Graham Wallis. Michel Factor wrote an installation tool for remotely configuring NetDispatcher at remote sites. Richard King also implemented an HTML configuration interface, and SSL affinity support. Ed Merenda and Rajat Mukherjee performed the laboratory performance tests on an SP/2. Steve Fontes and Chris Gage led the effort that produced the *"Interactive Network Dispatcher"*, an IBM product based, in part, on the NetDispatcher technology. John Tracey, Richard Neves, and Richard King ported NetDispatcher to Windows NT. Eric Levy ported it to the 2210 and 2216 hardware routers, and Steve Fontes lead a team that ported it to Solaris. Rajat Mukherjee prototyped HA for an SP/2, and Eric Levy implemented fault tolerance using two 2110 routers. Anne Reidelbach and Kevin Bouck tested NetDispatcher with many combinations of heterogeneous servers and networks. Susan Hanis performed performance tests on heterogeneous clusters. Murthy Devarakonda, Bill Tetzlaff and John Tracey provided valuable comments on early drafts of this paper. Murthy Devarakonda, Dan Dias, and Jim Russell jointly managed the project. In addition to all the above, we also had valuable discussions with Donna Dillinberger, Stuart Feldman, Ajei Gopal, David Grossman, Sean Martin, Chet Murthy, and Scott Penberthy.

References

[APB96] Eric Anderson, David Patterson, and Eric Brewer. The Magic Router: An Application of Fast Packet Interposing, May 17 1996. Unpublished Report. University of California, Berkeley.

[AS92] C. Attanasio and S. Smith. A Virtual Multiprocessor Implemented by an Encapsulated Cluster of Loosely Coupled Computers. Technical report, IBM Research, 1992. RC18442.

[AS94] Clement Attanasio and Stephen Smith. Method and apparatus for making a cluster of computers appear as a single host on a computer network, 1994. United States Patent 5,371,852.

[AYHI96] Daniel Andresen, Tao Yang, V. Holmedahl, and O. H. Ibarra. Sweb: Towards a scalable world wide web server on multicomputers. In *10th International Parallel Processing Symposium (IPPS'96)*, Hawaii, April 1996. http://www.cs.ucsb.edu/Research/rapid_sweb/SWEB.html.

[Bri95] T. Brisco. DNS Support for Load Balancing. RFC 1794, April 1995.

[CB95] Mark Crovella and Azer Bestavros. Explaining World Wide Web Traffic Self-Similarity. Technical report, Boston University, October 1995. TR-95-015.

[Chu96] Chi Chu. IP Cluster Internet Draft, August 1996.

[CIS96] CISCO. LocalDirector. http://www.cisco.com/, October 1996.

[DCH$^+$97] Om P. Damani, P. Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. In *6th International WWW Conference*, Santa Clara, California, April 1997. http://www6.nttlabs.com/HyperNews/get/PAPER196.html.

[DKMT96] Daniel Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A Scalable and Highly Available Web Server. In *Proceedings of COMPCON*, March 1996.

[EF94] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC1631, May 1994.

[ELZ86] D. L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[GY93] Germán Goldszmidt and Yechiam Yemini. Evaluating Management Decisions via Delegation. In *The Third International Symposium on Integrated Network Management*, San Francisco, CA, April 1993.

[Hyd96] HydraWEB. HTTP Load Manager. http://www.hydraWEB.com/, 1996.

[IBM96a] IBM. *IBM Parallel System Support Programs for AIX: Event Management Programming Guide and Reference*, 1996. SC23-3996-00.

[IBM96b] IBM. *IBM Parallel System Support Programs for AIX: Group Services Programming Guide and Reference*, 1996. SC28-1675-00.

[JH97] Rodney Joffe and Henry Heflich. Hopscotch Optimized Web Access, March 11 1997. http://hopscotch-sjc2.genuity.net/wpaper1-02.html.

[KBM94] Eric Dean Katz, Michelle Butler, and Robert McGrath. A Scalable HTTP Server: The NCSA Prototype. *Computer Networks and ISDN Systems*, 27:155–163, 1994.

[KK92] O. Kremien and J. Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Processing*, 3(6), November 1992.

[KL87] P. Krueger and M. Livny. The Diverse Objectives of Distributed Scheduling Policies. In *Proceedings of the 7th IEEE International Conference on Distributed Computing Systems*, pages 242–249, 1987.

[MM96] Ed Merenda and Rajat Mukherjee. Webstone performance mesurements of network dispatcher, 1996. Unpublished Internal Report.

[Moc87] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, November 1987.

[Mog95] Jeffrey C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical report, Digital Western Research Lab, October 1995. WRL Research Report 95/5.

[Res96] Resonate. A Case for Intelligent Distributed Server Management. Available from http://www.resonateinc.com, December 1996.

[Sch95] Roland J. Schemers. lbnamed: A Load Balancing Name Server in Perl. In *Proceedings of LISA IX*, Monterey, CA, September 1995.

[SL96] Mazahir Songerwala and Efrat Levy. Efficient Management of Multiple Web Servers, December 1996. Unpublished Project Report. Columbia University.

[Sta97] William Stallings. Viewpoint: Self-similarity upsets data traffic assumptions. *IEEE Spectrum*, January 1997.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[TS95] Gene Trent and Mark Sake. WebSTONE: The First Generation in HTTP Server Benchmarking. http://www.sgi.com/Products/ WebFORCE/WebStone/paper.html, February 1995.

[Wal96] Edward Walker. pWEB - A Parallel Web Server Harness, 1996. http://phobos.nsrc.nus.sg/STAFF/edward/pweb.html.

[YCE+97] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, and David Culler. Using Smart Clients to Build Scalable Services. In *Usenix 97*, January 1997.

[YHK96] Heon Y. Yeom, Jungsoo Ha, and Ilhwan Kim. IP Multiplexing by Transparent Port-Address Translator. In *USENIX Tenth System Administration Conference (LISA X)*, Chicago, IL, USA, September 29 - October 4 1996.