Scaling Internet Services by Dynamic Allocation of Connections

G. Goldszmidt, G. Hunt
IBM T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY
USA
{gsg, gdhh}@watson.ibm.com

Abstract

Network Dispatcher (ND) is a software tool that "routes" TCP connections to multiple TCP servers that share their workload. It exports a set of virtual IP addresses that are concealed and shared by the servers. It implements a novel dynamic load-sharing algorithm for allocation of TCP connections among servers according to their real-time load and responsiveness. ND forwards packets to the servers without performing any TCP/IP header translations, consequently outgoing server-to-client packets are not handled, and can follow a separate network route to the clients. Its allocation method was proven to be efficient in live tests, supporting Internet sites that served millions of TCP connections per hour. This paper describes the load management features of ND.

Keywords
Internetworking, Traffic Management

1 Introduction

As TCP/IP traffic increases, many Internet sites are often unable to serve their workload, particularly during peak periods of activity. This paper describes the load management of ND, a tool which enables TCP/IP server clusters to handle millions of TCP connections per hour. ND shares a collection of *Virtual* IP Addresses *(VIPAs)* with the cluster servers. Each VIPA is shared by the ND host and a *"Virtual Encapsulated Cluster" (VEC)*, a collection of (heterogeneous) servers that provide the same function and serve equivalent content.

For example, a VEC may consist of 10 hosts sharing the same VIPA, each accepting TCP connections on port 80 and delivering equivalent content. When a Web browser accesses a page, it may open several concurrent TCP connections, each of them may be routed to another VEC host.

ND 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.

The service time and the amount of server resources and network bandwidth consumed by each request varies widely. To address these workload characteristics, ND 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.

At the main Web site for the 1996 Summer Olympics, a single ND supported 4 VECs on some 50+ SP/2 nodes [5]. The 1998 Wimbledon Web site, handled over 150000 connections per minute, with negligible latency overhead. ND was used to implement several large scale Web sites, with Ethernet, Token Ring, FDDI, and ATM networks, supporting heterogeneous servers. We have found that greater performance and scalability can be obtained by configuring parallel and hierarchical sets of NDs.

In the event of a ND failure, its VIPA addresses can be transferred to another backup ND, by flushing the ARP cache on the corresponding IP routers. Complete recovery without (or with minimal) loss of active connections, is implemented by maintaining a shadow ND host that keeps track of all existing connections and other state.

The focus of this paper is the operational management of the ND, and its load management component. A more detailed description of all the components of ND, is given in [6]. Reference [8] includes a preliminary evaluation of ND's performance in the context of HTTP traffic, and other features, including high-availability, support for nonlocal server sites, and client affinity. Reference [5] describes some of the challenges of building, operating and managing a large distributed system using ND. Reference [4] describes the IBM *eNetwork Dispatcher* product, which is based on this work.

The rest of this paper is organized as follows. Section 2 describes how ND is used. Section 3 describes the Manager and its dynamic feedback load sharing algorithm, Section 4 describes the configurable load metrics index. Section 5 describes some alternative approaches to solve the scalability problem, and Section 6 concludes.

2 Configuration and Use

One of the most useful features of ND is the ability to dynamically change all configuration information. This enables system managers to add and remove services and servers without interruption. In particular, it enables the dynamic upgrade of content and software at the servers. ND provides a single point of control for the allocation of cluster resources, for collecting real-time logs, and monitoring the current performance of the cluster. As it dynamically monitors the performance of the cluster, it automatically removes servers that are not performing properly.

The administrator of ND defines the VIPAs, ports and servers of each VEC. The VIPAs become known via DNS and ARP. Each VIPA that a server supports must be known as an alias on an interface so that the local TCP stack will accept the corresponding incoming IP packets. However, the servers may not export the VIPAs via ARP, since that would create LAN conflicts. One method to do this is to give the alias to a non ARP-exported interface, like the *loopback* interface, (e.g., by performing `ifconfig` commands). Notice that ND, in contrast with other tools, does not require the installation of any software nor O/S upgrades at the servers.

2.1 Network Configuration

ND can be configured within one or several networks. Figure 1 shows a typical 2-network configuration of ND. The *internal* network connects ND to the servers (S1, S2, S3) and is used for ND-to-server packets. The *external* network is used for both incoming client-to-ND packets and for outgoing server-to-client packets. The client-to-ND 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, the external network could be a Token Ring where servers forward their traffic, and the packets could arrive into ND from an ATM link. In Figure 1 ND 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. The VIPA 129.34.129.8 is an alias to the loopback interfaces of the 3 servers.

2.2 Forwarding Packets

For each IP packet that represents a new TCP connection request, ND chooses a server from the target VEC, using a *Weighted Round Robin (WRR)* algorithm. The first and subsequent client-to-server IP packets for this TCP connection are forwarded to the corresponding server. Outgoing server-to-client packets do not need to flow through ND but may follow a separate route, as illustrated by Figure 2. Bandwidth utilization can be more efficient by allowing the server-to-client IP packets to follow a separate route through

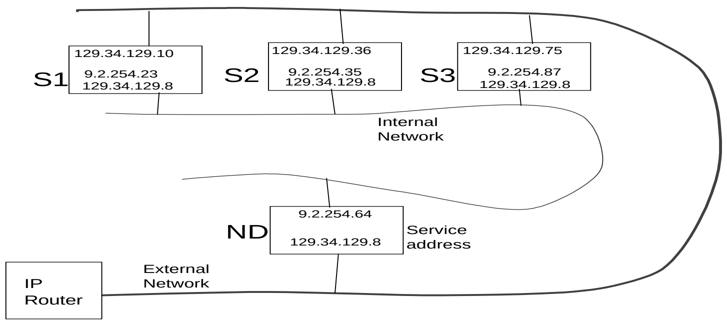

Figure 1: Sample network configuration for 2 networks.

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 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. When a Web server delivers multimedia data, the ratio of server-to-client outgoing bytes to client-to-server incoming bytes is very large. ND processes only the incoming packets. In contrast, header-translation tools must process both incoming and outgoing packets. Additional details of the WRR and the packet forwarding method are presented in [6].

Filtering Packets. ND 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, ND can be configured to only allocate connections destined for TCP port 80, and discard all other packets.

Quiescing, and Co-location. ND also supports *quiescing* servers, that is marking them as inactive. When a server has been marked as quiesced, the currently active 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 ND. This is very useful for low workload periods, during which ND may be configured to allocate most connections locally.

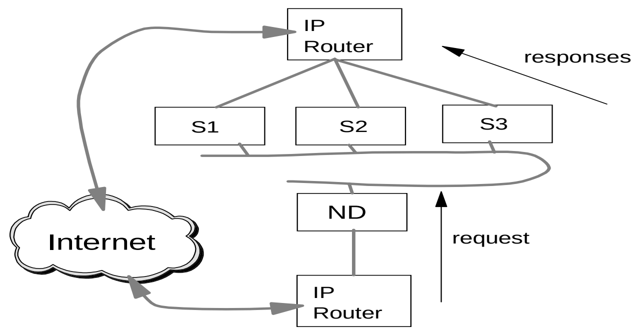

Figure 2: Example of ND Traffic

3 Managing Load in ND

The Manager component of ND 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 3 illustrates the relationships between the Manager, Executor, Advisors, and servers. *Advisor* processes collect and 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.

3.1 Dispatching TCP Connections

Remote clients, such as browsers, use TCP connections 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. For example, it can depend on the type of service being provided, the specific content involved in each request, or the current network bandwidth available. Some "heavy" requests perform long transactions that involve 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.

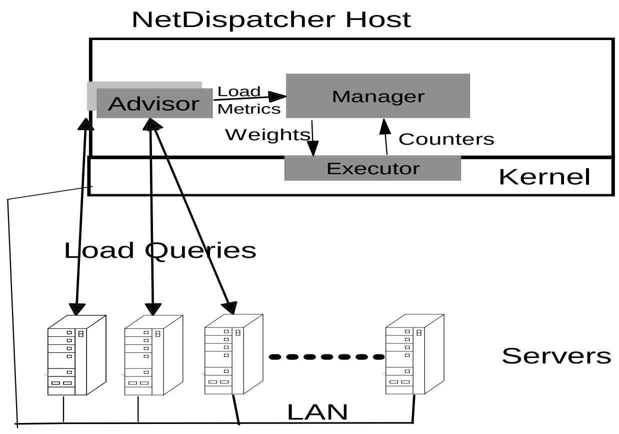

Figure 3: The ND Manager

This imbalance in request processing time often causes skewed utilization of the server cluster. For instance, consider an application that sequentially opens 4 concurrent connections to retrieve 4 graphic files, A, B, C, and D, one of which (D) is very large. Assuming 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.

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. Studies of TCP/IP traffic (e.g., [16]) show that surges occur in waves, where long periods of little traffic are followed by intervals of heavy usage. These studies also indicate that traffic peaks tend to cluster. 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 [2]. 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 ND to more evenly utilize the cluster resources. Implementing this in ND has the advantage of being transparent to both clients and servers.

3.2 Load Sharing with Dynamic Feedback

ND needs some guidance in order to allocate the TCP-connections in a way that utilizes the cluster resources efficiently. *Load-balancing* and *load-sharing* [3] are two techniques for improving performance by using several servers. Load-balancing strives to equalize the servers workload, while load-sharing attempts to smooth out transient peak overload periods on some nodes [11]. Load-balancing strategies typically consume many more resources than load-sharing, and the cost of these resources often outweigh their potential benefits [12]. The workloads that we observed at several highly loaded Internet sites were characterized by having many very short transactions and a few long ones. For this type of workload, ND used a load-sharing method that proved to be very efficient, and also achieved a relatively uniform distribution of the workload.

ND 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. 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 $\forall i \neq j,\ W_p(S_i) \leftarrow 0$, $W_p(S_j) \leftarrow 1$. 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 $(W_p(S_i) \leftarrow 0)$ before performing some maintenance on it, or after discovering a problem.

4 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. These metrics should be computed and gathered in a consistent manner for all the relevent servers of a VEC.

4.1 Metric Types

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 ND host. Host metrics are computed at each server host, and forward metrics are computed via network interactions between ND and each server host. These metrics could be further combined to provide a *health index* [7] for network management purposes.

4.1.1 Input Metrics

Input metrics provide a current estimate of the state of the VEC servers, as seen from ND. These metrics are derived from the counters and gauges that ND's Executor collects. 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.

4.1.2 Host Metrics

These metrics are computed at the server host, and represent some measurement of the load on some resource. The total number of active processes, or the total number of allocated mbufs in a given server are examples of host metrics. Notice that if the servers are heterogeneous, host metrics must be normalized. Host metrics are typically computed at each host server by an *agent* process that executes command scripts. These scripts return numerical values that are reported to a corresponding *Advisor* process at the ND 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 "disabled" 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 example, for the Olympic Games Web site [5], ND used configurable Advisor processes to collect host metrics, such as "number of active processes", for a set of homogeneous servers. The string, " `ps -ef | wc -l` ," together with a time value representing the period between computations, and a list of hosts and other configuration values were given to an Advisor. The Advisor sent the command string and its configuration parameters to all the corresponding host agents.

An advantage of this method is its 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.

4.2 Forward Metrics

Forward metrics are computed by sending messages from ND to a specific host service. For instance, the time required to retrieve an HTML page from a Web server to the ND host is a forward metric. An HTTP Advisor at the ND 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 disabled, and hence no more new connections of this type should be forwarded to it.

4.3 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. For example, a network manager 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 ability to tailor the LMI particularly useful in situations where the workload and the content of the servers changed frequently.

4.4 Weights Computation

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. For instance, one parameter is a smoothing factor that is 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.

The main loop of the WCF computes the aggregate metrics for all the executing servers by combining all the weighted metric instances. For each executing VEC server it computes its current weight and load proportions relative to the other servers. The input parameters to WCF include the above aggregates and ratios, the medium weight for the current port, the current weight of each server, the load metrics (and corresponding weights) of each server, and a configurable smoothing parameter.

In addition to the above metric types, network administrators can define arbitrary new metrics to be considered in a similar fashion for management of the connections. These metrics can be used to enforce any desired allocation policies.

4.5 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 ND. In other words, the delay in the feedback loop is often too long compared with the average transaction size to enable effective load balancing.

The load sharing algorithm was shown to be sufficiently sensitive to host overload, e.g., for 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 an 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 a site, as requests to that host may stay queued up for long periods of time. Marking the hosts as quiesced prevents incoming request from hitting temporarily unresponsive servers. In general, a site with a relatively stable and predictable workload may define a static LMI. However, for sites that have frequent workload variation, it is very useful to dynamically modify the LMI. Ideally, this could be done automatically by some automated learning mechanism.

5 Alternative Approaches

In [6] we describe in more detail the main characteristics of the following methods, and compare them to ours. In an independent evaluation [15], ND was shown to outperform other tools. We classify these alternative methods into the following categories:

Client-based choice of server by the end-user or implicitly by the software. It is not possible to ensure that this methods will produce a fair distribution of requests across the servers, as they don’t take into consideration the current load or availability of the servers.

Splitting each TCP connection into two TCP connections. The main disadvantage of this method (e.g., [18]) is its performance, in terms of both higher 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 [17].

DNS-based methods. These methods (e.g., [10]) enable clients to implicitly choose a server. Mogul observed that DNS-based techniques cannot provide linear scaling for server performance at peak request rates [13]. Caching of IP addresses creates skews the distribution of requests. Also, DNS solutions are very slow (or unable) to detect server failures and additions of new servers. A detailed analysis of many of the problems related to DNS-based solutions is presented in [6].

Forwarding of IP packets, source-based or by LAN broadcasting. For example, *Convoy* [14] forwards every client request to all the (NT) servers, which then filter the requests. Its scalability is limited by the throughput of the slowest device/interface card in the cluster.

Packet forwarding with TCP/IP header translation. This is the main method used by commercial load balancers, e.g., [1, 9]. There are several disadvantages of this 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.

6 Conclusions

Popular Internet sites need to scale-up to serve their ever increasing TCP/IP workload, particularly during peak periods of activity. ND supports the sharing of virtual IP addresses by several servers, and properly distributes the workload among them. Load allocation decisions are made in real time, as each request arrives, based on the current state of the servers. Dynamic load sharing enables efficient allocation of computing resources and reduces the service time of most requests. Configurable load metrics evaluated in real-time are necessary to provide customized feedback for the workload of each site.

Forwarding incoming client-to-server TCP packets unchanged is more efficient and scales-up better than alternative TCP header-translator methods. 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 ND. 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. The dynamic configurability supported by ND has been shown to be a very useful for network managers.

From our experiences of using ND to build large scale TCP/IP service sites, we concluded the following: First, avoiding IP header translations (as done by other tools) has many significant performance advantages. 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 the workload of each site. Fourth, a load balancing is an excellent instrumentation point for real-time and off-line monitoring of the cluster services.

References

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

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

[3] 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.

[4] Chris Gage. IBM eNetwork Dispatcher Version 2.0 Olympic-scale TCP/IP Load-balancing and availability. White Paper (unpublished), June 1998.

[5] Germán Goldszmidt and Andy Stanford Clark. Load Distribution for Scalable Web Servers: Summer Olympics 1996 - A Case Study. In *Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, Sydney, Australia, October 21-23 1997.

[6] Germán Goldszmidt and Guerney Hunt. NetDispatcher: A TCP Connection Router. Technical report, IBM Research, Hawthorne, New York, July 1997. RC 20853.

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

[8] Guerney D. Hunt, Germán Goldszmidt, Richard King, and Rajat Mukherjee. Network Dispatcher: a connection router for scalable Internet services. *Computer Networks and ISDN Systems*, 30(7):347–357, April 1998.

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

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

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

[12] 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.

[13] 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.

[14] Valence Research. Convoy Cluster Software. White Paper (unpublished) http://www.valence.com, 1998.

[15] J. William Semich. New Web Clustering Systems to Improve Server Response. *Web Week*, 3(2), January 20 1997.

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

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

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

Biographies

Germán Goldszmidt is a Research Staff Member at IBM's T.J. Watson Research Center, where he has been working since 1988. Dr. Goldszmidt's research interests include networking, distributed systems management, and their applications to e-commerce. In 1990 he started graduate studies at Columbia University, where he developed a network management framework, MbD. He received his Ph.D. in Computer Science from Columbia University in 1995. Before joining IBM, he was a research assistant at the Technion, Israel Institute of Technology, where he developed a debugger for distributed programs. He received his M.Sc. (1988) and B.A. (1985) from the Technion, all in Computer Science.

Guerney D.H. Hunt is a Research Staff Member at IBM's T.J. Watson Research Center in Yorktown Heights, NY. Dr. Hunt's current research interest include middleware and infrastructure to support ubiquitous computing. He received his B.S. in Mathematics from Michigan Technological University in 1973, an M.S.in Computer Science from Cornell University in 1975, and a Ph.D. in Computer Science from Cornell University in 1995. From 1975 to 1981 he worked for the NCR corporation in Ithaca, NY. From 1981 to 1990 he worked for the IBM corporation in Endicott, NY. Since 1995 he has been at the IBM T.J. Watson Research Center and has worked on Internet technologies, systems technologies, operating systems, and middle-ware.