The Application Gateway System

Final Report

1997

Introduction

The Application Gateway System (AGS) is a distributed server system consisting of multiple heterogeneous servers that appears as a single high performance system on the Internet. It can provide stand-alone services or access back-end capabilities such as databases or repositories.

Clusters of server machines, or "server farms," are often used to provide a single service on the Internet. This is a cost-effective way to increase performance and availability of the service. The challenge is to set up the cluster so that the end-user is transparently connected to one of the servers in the cluster. Some of the commonly used techniques are round robin DNS, packet rewriting, and custom client software. Each technique has its strong and weak points, which we will discuss below.

We have designed and implemented a new approach, using probabilistic assignment of jobs to servers based on currently observed load figures per server. Our design addresses some of the shortcomings of existing solutions by providing an adaptive and scalable load balancing mechanism without a central point of failure. This research effort was supported by the National Science Foundation (NSF) and the National Library of Medicine (NLM).

Goals of the AGS Project

We set forth the following goals for the project, in part based on specific requirements from NLM.

- End users should not need to know that multiple servers make up the service.
- Reconfiguration of the service should be easy and, where possible, automatic. This includes adding servers to the cluster, taking servers down for maintenance, and dealing with failures such as crashed servers.
- The overhead introduced by the load balancing should have minimal effect on overall performance, and the AGS should scale well.
- The AGS should support heterogeneous clusters: the machines making up the server farm should not need to have the same capacity or run the same operating system.
- It should be possible to use a single cluster to provide multiple independent services (e.g., HTTP, FTP and Z39.50). Not every machine in the cluster needs to run a copy of each service.

Review of Existing Approaches

Before describing the AGS design, we review three existing approaches:

- Round-robin DNS
- A custom client solution used by Netscape
- A commercial packet rewriting device, Cisco's LocalDirector product.

Round-robin DNS

Round-robin DNS is a widely used modification to the standard DNS name resolution service. It maps a single name to a list of addresses, returning a different item from the list for each subsequent resolution request (hence the term "round-robin"). This is a simple change to the DNS server software, and requires no modifications to the DNS client or any other client software.

Round-robin DNS requires that the servers in the farm have comparable capacity, since by design it causes each server to receive the same share of the requests. This makes it difficult to use a heterogeneous server farm, and causes problems when servers crash unexpectedly. Other problems are caused by the extensive caching built into the DNS system. A single response may be shared by numerous clients (perhaps all clients on a large company's intranet),

causing uneven load on the server machines, and propagation of changes (e.g. to withdraw a crashed server) is slowed down by the cache entry lifetime.

A refinement to round-robin DNS solves some but not all of these problems. A second level of indirection is added using the ability to assign multiple (virtual) IP addresses to a single (real) machine.

The round-robin DNS service can map a name to a list of virtual addresses, which never needs to change. Configuration changes are implemented by changing the mapping from virtual addresses to machines; this can be done quickly without worrying about cache entry lifetime. The scheme can account for machines with different capacity by assigning a more powerful machine a larger share from the set of virtual addresses. It also reduces (but does not completely eliminate) the imbalanced load caused by DNS caching. The scheme does not, however, automatically adapt itself to changes in actual load or server crashes.

A Custom Client Solution

The Netscape browser has a built-in mechanism that rewrites the hostname when it detects that a URL references Netscape's own web site. It substitutes a randomly chosen host name from a precompiled list and then processes the URL as normal (without changing the URL as seen by the user). This effectively eliminates the caching imbalance, at the cost of fixing the mapping in the client software, which cannot be changed once distributed. Of course, most configuration changes can still be implemented by changing the DNS mapping in Netscape's DNS server. This solution does nothing for the other disadvantages of round-robin DNS. The major drawback for general application of this mechanism is of course that it requires a custom client for each service.

A Packet Rewriting Device

Cisco's LocalDirector product is an example of a packet rewriting device. All traffic to and from the service cluster flows through the device, which is similar to an IP level bridge. All traffic going into the cluster destined for a specific virtual IP address is intercepted and rewritten to go to one of the servers in the cluster. Corresponding traffic coming out of the cluster is rewritten to appear to come from the same virtual address. Other traffic goes through the device unchanged. The LocalDirector monitors the performance of the servers in the cluster, primarily by monitoring how quickly they respond to certain types of traffic, and uses this information to select a server on a per-connection basis.

Each packet must be modified and its checksum must be recomputed (see [RFC 1631]), and all traffic to and from the cluster must flow through the device. This means that the device must deal with a much higher volume of traffic than individual servers in the cluster, and at the same time forms a single point of failure in an otherwise redundant and replicated system. To compensate for this, high-end hardware is used in combination with a reliable dedicated real-time operating system. For additional reliability, a second unit can be wired to act as a hot standby (but not to increase throughput).

There are other packet rewriting techniques that avoid the need for all traffic to go through a single device. However, these require custom OS kernels and applications, and don't completely avoid the central point of failure problem.

The AGS Approach: Rate.d

The AGS uses an adaptive load-balancing algorithm, called rate.d, which avoids introducing a single point of failure. The algorithm adapts quickly to changes in configuration or actual load of the server machines.

The rate.d algorithm requires custom client code but is more flexible than Netscape's custom client solution. Compared to round robin DNS, it provides fast, automatic adaptation to changes in load or configuration. Compared to LocalDirector, rate.d has no central point of failure and can use more direct measurement of server load.

Quick Overview

The rate.d algorithm has the following properties:

- Fully distributed—no central point of failure. Each server acts as a backup for all other servers.
- Adapts quickly and automatically to crashed, deleted or added servers.
- Adapts quickly and automatically to dynamic changes in individual server load.
- Configurable so that higher performance servers take a larger share of the requests.
- Low internal traffic rate—no intra-cluster traffic is needed to handle each request.

- Probabilistic approach avoids most pitfalls caused by deterministic server selection.

In addition, the AGS rate.d implementation supports multiple independent services run on overlapping subsets of the cluster, and is integrated with the AGS management system, which supports convenient secure remote configuration monitoring and control.

The Rate.d Load Balancing Algorithm

The AGS load balancing algorithm works by running a daemon program, the rate.d daemon, on each server machine in the cluster. Using IP broadcast, the client sends a single packet containing a request for service to the cluster. The rate.d daemons on all participating servers receive and evaluate the request.

The algorithm selects a single server from the cluster to handle the request, using a weighted lottery (described below) to balance the load. The selection is made by each daemon independently, but each daemon (usually) makes the same selection. The selected server sends a unicast response to the client. Once the client receives the response, it initiates the actual service connection to the server machine indicated in the usual way; for example, by opening an FTP or HTTP connection.

Each daemon measures and broadcasts its load to the other daemons at least once a second. Each daemon monitors these broadcasts to keep a loosely consistent view of the load on all servers. The load information is used to create a set of lottery tickets so that the number of lottery tickets for a particular server is inversely proportional to its most recently observed load, scaled by a factor that takes the static capacity of the machine into account. Thus, more powerful machines get more tickets, everything else being equal. One ticket is randomly picked as the winner. A hash of the request packet is used as the sole source of randomness for the lottery; the history of previous lotteries held has no effect. This ensures that lost client requests do not affect the stability of the algorithm.

When the daemons have exactly the same load information, they all pick the same winner, and when they have approximately the same information, they are likely to pick the same winner. We assume that the load of a machine remains the same until another update is received. This assumption, together with the properties of the lottery, means that losing an occasional broadcast update from another daemon does not greatly affect the lottery outcome. The same is true when updates and requests are received in different orders by different machines. On the other hand, when no updates are received from a particular machine for a longer period of time, that machine is deemed to have crashed and is no longer considered as a candidate.

Occasional inconsistencies between machines will cause a duplicate response or no response to be sent. Duplicate responses are dealt with by the client, which ignores all responses after the first. The client also deals with missing responses by timing out and resending the request—this is the only way to guarantee a response in the case that the initial request or the only response was lost. Such a timeout has to be conservative, however, in order to deal with varying and unknown network latencies: The client may be connected to the service cluster from far away on the Internet. Therefore, the rate.d algorithm provides a fallback mechanism.

As part of their regular update broadcasts, each rate.d daemon also transmits a list of the request IDs to which it has recently responded, and the other rate.d daemons track this information. For this purpose, the rate.d client library assigns a 64-bit unique identifier for each request. When a server that should have responded to a particular request fails to do so, the daemons on the other servers remove all tickets for the particular request from their lotteries, and draw another winner. If necessary, this process is repeated at regular intervals. The algorithm always terminates: Eventually each daemon will determine that it is the winner.

This fallback mechanism takes care of three different sources of inconsistencies between lottery outcomes: differences in relative arrival times between requests and update broadcasts, loss of request packets on some machines, and partial or total loss of update broadcast packets. Occasionally, a second lottery is held and a second response sent unnecessarily; in this case the client would just ignore the second response.

The Client Side

The client side of the rate.d interaction is fairly simple. An API for C programmers has been developed as part of this project to make it easier to integrate rate.d allocation into an application. Some useful functions are also provided for Python programmers. The definitive document on the interaction of clients and the rate.d daemon is “The Rate.d Distributed Load Balancing System” [Rate.d].

The client needs to perform a few basic operations in order to use the services of an AGS. The implementation of the client API for C programmers takes care of these operations, but an overview is in order. The client is required to know two pieces of information: the IP address of the AGS and the service identifier used to specify a specific service. Some services may require additional, application-specific information to be provided as well. The service identifier is a 32-bit integer. A request message is sent to the AGS at the IP address at an assigned port via UDP. The client then must wait for a response from a server on the AGS for information needed to contact a specific server; the response will include an IP address, a port number, and, if needed, application-specific information. The client can use this information to make a connection to the service, at which point no further special consideration need be made for working with the AGS.

Architectural Limitations

There are some fundamental limitations that should be understood before adopting the rate.d architecture.

Clients need to be modified in order to participate in the rate.d negotiation.. This is due to the use of an explicit request for service. To support unmodified clients, proxies can be provided which perform the rate.d request for service and establish the application connection on behalf of the client, redirecting traffic between the client and application once the connection is established. Such a proxy can be located at any point, and need not be co-located with either the client or server farm—a logical place would be on one of the AGS server machines though. This is discussed in more detail in "Rate.d Distributed Load Balancing System" [Rate.d]. In contrast with LocalDirector (mentioned above), a rate.d proxy can be an application-level service. Since a proxy adds extra round trip delays and is a single point of failure, it should only be used in order to support legacy clients that will eventually be phased out.

Because of the request for service, the protocol increases the total latency clients experience. The increased latency is caused by the time taken for the round-trip to the server farm consisting of the request message from the client and the response from a server.

Rate.d has scaling limits. The current design requires that each server to independently determine the acceptance of each request. This imposes a moderate amount of computation on the CPU at each server, reducing the total available computing power available for servicing client requests. It also imposes more internal communications than would use of a single leader that makes all the determinations. The advantage of the current design is a higher level of availability in the face of failure; there is no single point of failure.

AGS Architecture

In a typical configuration, the AGS is composed of two or more servers on a dedicated LAN segment separated from other networks by routers, as shown in Figure 1. In the figure, the router on the right separates the AGS network from the Internet. The router on the left separates the AGS network from an internal network. The internal network may house database hosts which are accessed by AGS servers on behalf of client requests, such as NLM's MEDLINE database, and workstations on which the AGS management system, described below, may run.

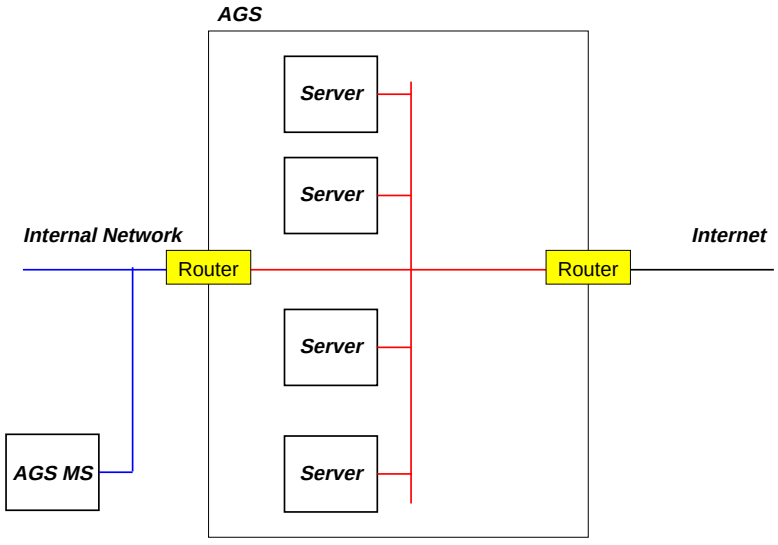

Figure 1. Basic AGS Architecture

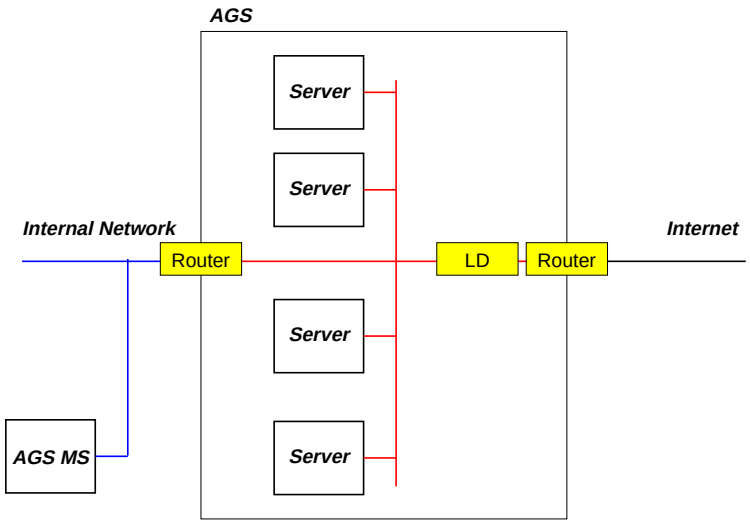

Figure 2. AGS Architecture with Cisco LocalDirector

When the Cisco LocalDirector is used in place of rate.d, it is placed between the router to the Internet and the servers as shown in Figure 2.

As shown in Figure 3, each AGS server runs a Supervisor process and a rate.d process. The supervisor is responsible for accepting commands from the AGS management system (see below), including commands to start and stop services and rate.d.

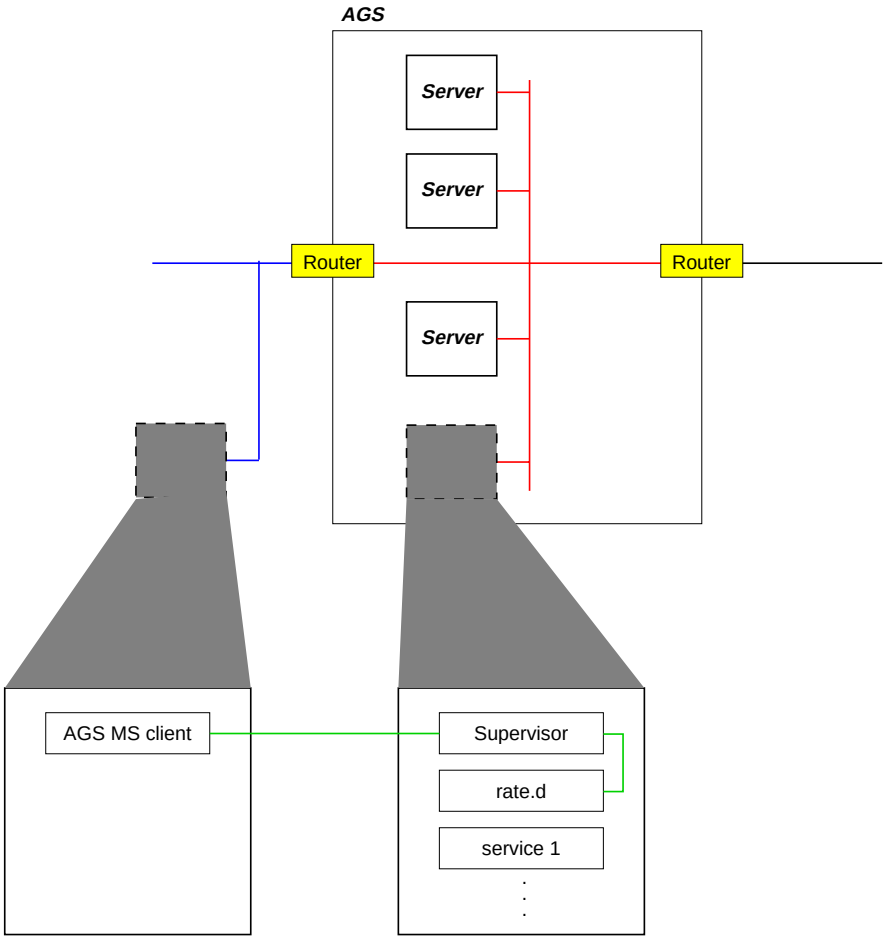

Figure 3. Processes in the AGS

Security Issues

No extensive security analysis of the AGS software has been made. However, the architecture (see Figure 1) has been designed with security in mind. In particular, the design sketched above, with one router separating the AGS from the Internet and a second router separating it from an internal network, makes it possible to use the AGS as a firewall, using appropriate packet filters in the two routers.

In addition, the AGS management system uses Kerberos to authenticate the to the AGS management system. The security of the management system is discussed in more detail below.

The AGS Management System

The Management System (MS) can securely query and control the operation of the Application Gateway System across the network. Server-side components interact directly with the rate.d daemon, handling requests from client-side components that send AGS management requests. These requests include commands controlling the activation of the AGS server and of particular services on each host, and querying and setting of the AGS server operating parameters. The client-side and server-side components communicate with each other using the AGS Control Protocol. These communications are conducted over communication channels that have been authenticated using Kerberos.

Server-side MS Components

The primary server-side MS component is a daemon called the Supervisor. The Supervisor starts and stops the rate.d process. The Supervisor incurs low system overhead and is intended to be started at boot-time and left running perpetually.

The Supervisor is responsible for user authentication and for authorizing user privileges, passing valid requests on to rate.d. Kerberos is used for user authentication; access control lists are used for specifying user privileges.

Client-side MS Components

The client side of the MS is implemented as a library and a command line tool. The command line tool can issue commands to specific AGS hosts or to the entire collection at once. Both the library and the command line tool are structured for convenient use as components in construction of other MS clients; for example, graphical status monitors and controllers.

AGS Control Protocol

The AGS Control Protocol is used by the server and client components of the MS to communicate commands and responses. The protocol implementation is easily extensible.

MS Operations

The MS currently supports the following operations:

- Startup and shutdown of the rate.d daemon process.
- Enable and disable rate.d load balancing.
- Shutdown the Supervisor process (once shut down, it has to be restarted manually).
- Set various parameters that control the rate.d algorithm.
- Re-read the shared configuration file.
- To request various status reports, such as a list of configured services and their current state, load statistics, or a list of pending jobs.

MS Security

MS security and authentication is implemented using the Kerberos authentication system. This allows the Supervisor to be sure that administrators are who they claim to be.

Authorization is then provided and enforced by the Supervisor based on access control lists specifying the user IDs of the AGS administrators and the set of privileges that each administrator is given. Storage of the access control lists depends on the implementation of the Supervisor. In the prototype AGS, the access control lists are stored in text files protected by file-system permissions. Production systems should take additional steps to ensure the integrity of the access control lists as well as other configuration information. Encryption or other techniques may be useful in supporting integrity of configuration files for the AGS, or an alternate approach to configuration persistence may be used. One possible solution may be offered by the Application Configuration Access Protocol [ACAP].

The AGS Test Environment

Introduction

We have identified several different approaches to load balancing among machines in a server farm. How do we know which approach works best? How can we compare rate.d against other approaches? We needed a testing environment to evaluate the various load balancing algorithms under varying but controllable loads.

What did we test and what did we measure? Although there are many parameters that we could measure, one of the most useful is a comparison of the number of job requests sent to the AGS, versus the number of transactions completed. For example, we might set up a test where the AGS received an aggregate rate of 100 HTTP requests per second (RPS), and we would like to know how many successful HTTP transactions the cluster completed. This let us estimate how well a given approach might perform under realistic loads that highly popular Web sites routinely handle. For example, CNN's site might see 150 million hits or more on a big news day.

We wanted to test both sustained load levels and bursty load levels, where bursts of activity can greatly exceed the capacity of the AGS. How well do the various approaches shed load when they are under overload conditions?

The workloads requested should also be realistic. In the case of an HTTP server this means that they should include a mix of small and large files, and various CPU intensive CGI scripts. The distribution of requests can be gathered from replay logs of popular Web sites. We also needed to simulate request profiles from the real world—where there are a very large number of independent clients spread throughout the Internet—using only a limited number of clients on a test LAN. Our test harness maximizes the limited client resources available to load the server farm, and attempts to generate a load that simulates conditions on a WAN.

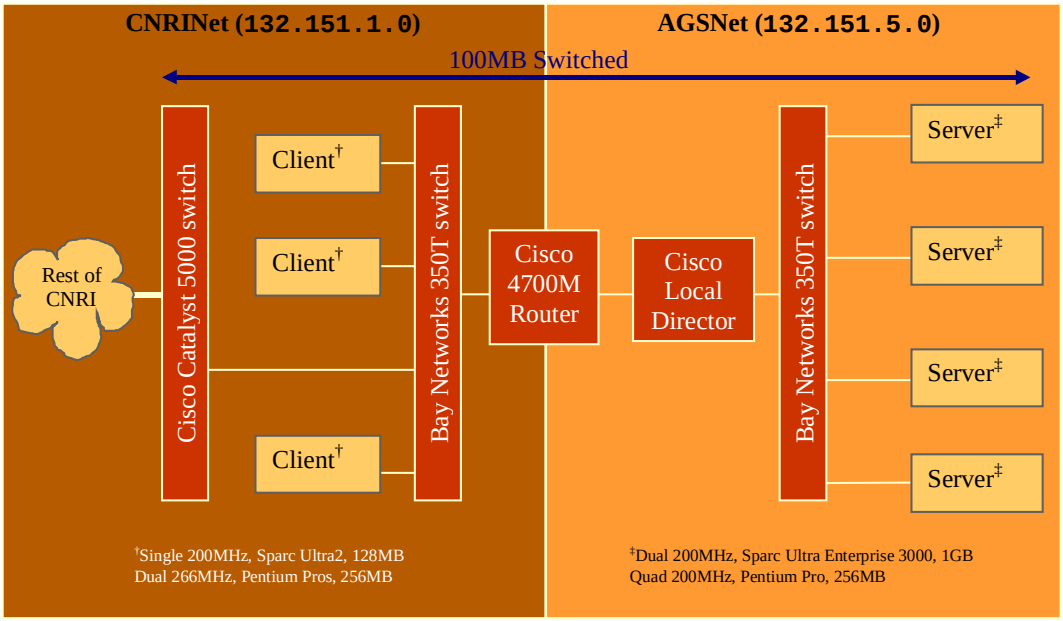

Figure 4. Network architecture of the AGS test environment

Test Network Architecture

Figure 4 illustrates the network architecture of the AGS test environment. The goal is a fairly isolated network where the underlying network hardware has a minimal effect on the measurements. (We were only partially successful with this; see the discussion in the Results section below.)

Working from right to left in this figure, we start with the individual servers that comprised the AGS server farm. For the typical test, five servers were used: two dual CPU 200MHz Sparc Ultra Enterprise 3000 servers with 1GB of memory each, and three quad CPU 200MHz Pentium Pro servers, with 256MB of memory each. All servers were running Solaris 2.6 with the latest OS patch set for the appropriate hardware platform, and were equipped with 100Mb/sec Ethernet boards. The HTTP server used is Apache 1.3b3.

All the servers were connected to a single Bay Networks BayStack 350T auto-sensing, auto-negotiating 10/100 switch, with all ports configured to run at 100Mb/sec. One port on the switch served as an outbound connection to the rest of the network, connected to a Cisco LocalDirector 415 (original LocalDirector model number CA-LDIR) with two FastEthernet (100Mb/sec) ports.

The other port of the LocalDirector was connected to a Cisco 4700M router which routed between the AGS test network and the rest of the CNRI network. This router was then connected to a second BayStack 350T switch, also set at 100Mb/sec on all ports. To this second switch were connected most (but not all) of the client machines that participated in the test harness. The second switch was used to reduce the impact of the AGS testing on CNRI's main internal network. The client machines included three dual 266MHz Pentium Pros with 256MB of memory and the AGS development team's desktop workstations, single-CPU 200MHz Sparc Ultra 2 machines with 128MB of memory. All clients were running Solaris 2.6 with the latest patch set, and contained 100Mb/sec Ethernet cards.

This yielded a maximum of 13 clients that could be utilized to generate load during any particular test, although the exact number of clients per test varied.

The second 350T switch also had a connection to CNRI's central Cisco Catalyst 5000 switch, containing several 10Mb/sec ports, and a single 100Mb/sec module with 12 ports. Most of the AGS test client machines that were not connected to the second Bay Networks switch were connected to 100Mb ports on the CAT5000.

A significant benefit of this network design is that the majority of the test traffic was isolated from CNRI's operational network. In early test runs conducted before addition of the second 350T switch, all the client machines were attached to the CAT5000. The effect on CNRI's operational network was quite noticeable when AGS tests were conducted, and incidental activities on CNRI's operational network affected the test results. By configuring the network as described above, we eliminated most of the interference between test traffic and operational internal traffic, while maintaining the benefits of shared CNRI resources such as automounted software directories, NIS+, DNS, etc. This arrangement also allowed us to have primarily 100Mb speeds throughout the entire testbed. Although some occasional client machines may have had 10Mb connections to the testbed, this did not significantly impact the test results.

Finally, note that the Cisco LocalDirector was connected in series with the AGS cluster. This was the case even when using the LocalDirector in "pass-through" mode, where the tests did not address the virtual interface, but instead addressed the real servers on the back-end. Our testing indicated that running the LocalDirector in pass-through mode had little or no effect on the measurements.

The Scalable Client

The primary difficulty in measuring AGS capacity is that we were attempting to simulate the real world—where many independent clients are scattered throughout the Internet—with a relatively small number of clients connected to a LAN. The WAN vs. LAN differences alone cause effects that can skew naive load generating algorithms.

To alleviate these effects, we adopted the "S-client" or Scalable Client architecture introduced by [SClient]. This paper enumerates a number of problems that can occur when attempting to heavily load Web servers. The issues and solutions described can be applied to any TCP based service. The paper shows how attributes inherent in TCP, which provide important benefit during normal operation on a WAN, can conspire to limit the ability of a LAN testbed to generate realistic heavy (and bursty) load.

Two important effects that the S-client alleviates are TCP's *exponential retry*, and *receiver-livelock*. We'll describe these effects only briefly here; see [SClient] for more details.

TCP provides reliable connections, so when a client's attempt to connect does not succeed within a certain time frame, TCP will normally attempt to retry the connection a few times. These retries occur at successively longer intervals. For example, one implementation of TCP retries connections at 6 seconds, 30 seconds, and 75 seconds, and if this final retry fails, the connection itself fails. This retry pattern is called exponential retry. The typical client application usually blocks waiting for the connection to complete.

However, when a server is heavily loaded, it can spend all its resources processing incoming network packets (which usually have a high priority), only to discard them because there are no resources left for the application programs to process the packets. This state is referred to as receiver-livelock and it causes clients to generate new requests at a very low rate (assuming a typical test setup with a small number of clients sending requests rapidly in order to simulate a large number of clients sending requests rather infrequently). Because the server's TCP stack has a limited sized queue to contain incoming packets, it will eventually begin to drop connection requests from clients, forcing the clients into exponential retry.

The effect of this situation on a naïve LAN-based test harness is that client processes that are making requests to a server become throttled, because they are waiting for successful socket connections. Since the connections are made sequentially, and since exponential retry can cause the process to block for more than a minute on each connection attempt, the clients slow their request rate. Thus they are unable to generate a rate far greater than what the servers can handle -- as would be the case if thousands of unrelated clients on the global Internet were making simultaneous requests of the servers in parallel.

Two techniques are used by the S-client to avoid this problem. First, S-client avoids TCP's exponential retry by closing unconnected sockets before the first retry interval. Second, S-client opens sockets in non-blocking mode, and maintains a constant pool of connecting sockets. When a socket connection succeeds, it transfers the connected socket to another process, which performs the HTTP request/response cycle. This second process operates only on successfully connected sockets.

Should the socket not connect before the first TCP retry interval, the socket is closed and thrown away. Because the sockets are opened in non-blocking mode, many connecting sockets can be managed by the S-client at a time. Thus a single S-client process can generate a much higher overall request rates than a naive client.

Our AGS test harness implements the S-client architecture in a Python program. We used our S-client program to generate high loads, which can be metered for a target aggregate system request rate, with a random distribution on the intervals between individual requests. The amount of load a single client machine can generate is dependent on the capacity of the client. In our tests, the single CPU 200MHz Sparc Ultra clients can comfortably generate approximately 125 RPS, up to a maximum of about 150 RPS. Thus the total aggregate load we can place on the AGS is about 125 RPS times the number of clients in a particular test run. For a test run consisting of up to 13 clients, each running at about 125 RPS, requesting only small HTML files (i.e. no CGI), this rate was easily enough to overload the Cisco 4700M router, crashing it and rendering the test network unreachable (until the router was rebooted).

The AGS Test Harness

We needed to conduct a large number of test runs, generating a huge amount of test data. Our tests needed to log results such as client-side target request rates, response rates, server-side response rates, and HTTP logs, along with output from Unix processes such as snoop, vmstat, and iostat. Load-balancing processes such as rate.d produced additional log information.

All of this information had to be collated and plotted, and the individual tests needed to be as reproducible as possible. Of utmost importance was the requirement that tests could be run unattended, since many of the tests were run overnight or over a weekend. Each test run was nominally 3 minutes long. (We performed longer tests, but this seemed not to affect the results significantly.) In addition, start-up and cleanup for a test run takes about a minute. For example, a single test of three different load-balancing algorithms over request rates from 100 to 1500 RPS at 100 RPS intervals takes over three hours. There were many other varying factors to test.

As a side note, we quickly learned that many factors can completely change the overall behavior of the AGS. For example, accessing local files and logging to local disks instead of NFS partitions made a huge difference in the overall throughput of an HTTP server. Changes in router configuration, Apache configuration variables, etc. greatly affected the results. In general we tried to maximize the raw throughput of the system so that we could vary primarily the load-

balancing algorithm used, but this was challenging. We often found ourselves debugging a network problem, only to correct a parameter in the router that wound up invalidating all prior test runs.

To improve reproducibility, and so automated test runs could be performed from a single workstation, we developed a Python script-based AGS Test Harness infrastructure. The harness allowed a script to conduct all test activities. This includes configuration, set up, and initialization of all test participants (clients and servers), execution of the tests, and at the end of the run, shutdown of the servers and relocation of all relevant log files to a central location for collation.

We also developed tools for analyzing the generated data, including scripts that summarize individual client results, and generate the result plots shown in the results section of this report.

Results

We compared the performance of job assignment using rate.d with LocalDirector and with a simulated round robin assignment. We ran two sample HTTP workloads using the hardware configuration and test harness described in the previous section. Results from these tests suggest that rate.d and LocalDirector perform similarly for modestly compute intensive requests, discounting wide-area network effects that were not present in our test environment. Rate.d would cause higher latency in an operational environment because it adds an extra network round-trip before establishing a TCP connection.

The sample workloads were derived from the server logs for a Web site (http://the-tech.mit.edu/Shakespeare/) that was mirrored on each server's local disk. The Web site has several hundred individual pages and a large HyperNews discussion area, accessed via several CGI scripts written in Perl. The test harness replays server logs to generate the test loads. The first workload, Workload A, includes all access to the site. The second, Workload B, is the same with all accesses to CGI scripts removed.

In our tests, we have focused on workloads that have a relatively high CPU usage for some jobs, because those are the HTTP workloads rate.d is best suited for. For the simplest HTTP workload, requests for static files, the server does very little work per job, so the relative overhead of rate.d is substantial. Only in workloads with high compute-to-request ratios does the overhead become not significant. In workload A, approximately 10 percent of all requests are for pages generated by CGI scripts. At this compute-to-request ratio, rate.d performs well. In workload B, LocalDirector and round robin assignment perform better, because the work per job is small.

For Workload A, the cluster can sustain performance of nearly 250 requests per second (RPS)[1]. The CPU and memory requirements of the CGI scripts are the limiting factors. For Workload B, the cluster can perform far more work but the network becomes saturated at nearly 550 RPS. Because network effects dominated in Workload B, we report primarily on results for Workload A.

The next two sections present a detailed analysis of results for Workload A. Section 4 describes configuration issues specifically related to the load balancing strategy. Some limitations of the current implementation are described in Section 5. Section 6 briefly presents results for Workload B. In the final section, results of single-server-failure tests are presented.

Overview of Results for Workload A

The peak 5-server cluster performance is at approximately 250 RPS. At lower request rates, all three mechanisms (rate.d, LocalDirector, and round robin) deliver throughput within one percent of optimal. LocalDirector delivers the highest peak performance, but it is only marginally faster than rate.d. Once the client request rate exceeds the cluster capacity, rate.d outperforms LocalDirector and round robin. Figure 5 summarizes the throughput of all three mechanisms at request rates from 50 to 800 RPS. Figure 6 and Figure 7 report the average request latency and the number of timed out requests. The results are described in detail below.

[1] It is important to realize that the performance measurements we report here are for a specific workload. The absolute numbers mean nothing by themselves.

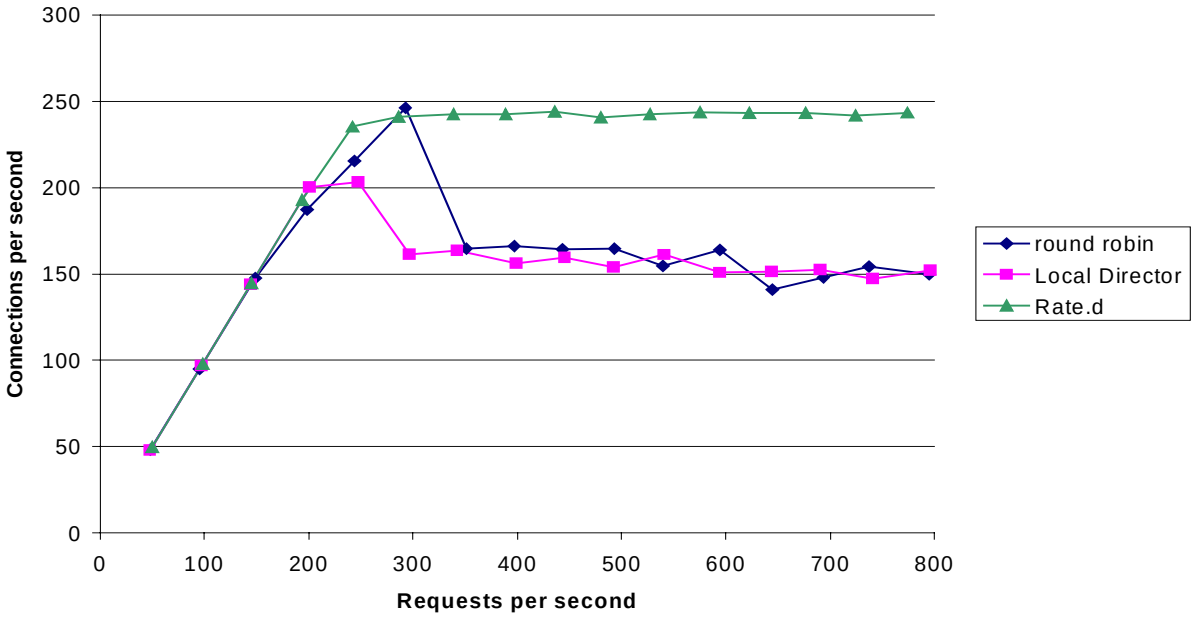

Figure 5. Throughput on a 5-server cluster

Throughput

We measured throughput by observing the number of TCP connections that were successfully completed by the test clients.[2] Throughput is reported in completed HTTP operations per second. The client-side measurements were informally verified by consulting the server-side HTTP access logs.

On workload A, each mechanism can deliver approximately 245 RPS. Each assignment system achieves nearly optimal performance as request rate scales up to the peak rate of 245 RPS. After the request rate exceeds 245 RPS, the behavior of each of the systems varies.

The measurements of peak load are complicated by frequent failures of several servers under round robin and LocalDirector job assignment; nearly every test run above 300 RPS resulted in a failure. The failures lower the average throughput for a test run, because the failed servers stop handling new requests. We have not made a conclusive examination of these failures, but we do describe their symptoms and possible causes and suggest why rate.d seldomly exhibits the same behavior.

The failures occurs on the three Intel servers. We suspect the failures are caused by memory contention, because the problem only occurs on the three servers that have 256MB of memory and never of the two Sparc servers, which have 1 GB of memory. The failed servers exhibit the following symptoms:

- The server stops performing useful work. The number of new HTTP requests that are completed is almost zero. At the end of the test run, the test harness does hard kills of all server processes. After a few minutes, the system appears to recover.
- The httpd error logs contains lots of error messages, including “Out of memory!” and “Child process still didn’t exit.” These messages appear to be caused by CGI processes that are started but never make any progress; apparently, the OS never executes any user code in the process.
- For rate.d tests, which fail much less frequently than round robin or LocalDirector, one rate.d process falls behind processing internal broadcasts. As a result, this rate.d process believes that no other server is accepting jobs and tries to accept all incoming requests. The rate.d process consumes an inordinate amount of CPU time.

[2] Some of the successful TCP connections delivered HTTP errors at the application level. These connections were considered successful because we used real server logs to drive the test clients; these logs contained requests that result in errors. In real-world operation, the server must deliver HTTP error messages to clients.

It appears that rate.d fails less often because of two side effects of the overall design and implementation. First, rate.d uses a UDP packet to do the initial request for service and response. Second, rate.d runs as a user-level process on each server. At very high request rates, the user-level rate.d process can not get enough CPU time to process every incoming request. As a result, the operating system drops some UDP packets before rate.d ever sees them. The system does less work to drop a UDP packet than it does to drop a TCP connection, so the UDP service acts as an inexpensive throttle on new job acceptance.

Latency

Latency is measured using a separate test client that reports that total time for an HTTP operation to complete. (The latency measurement client does not use the S-client time out mechanism that the throughput measurement client uses.) The test client makes approximately 60 measurements per test run, measuring time from the first packet sent, either the rate.d Request for Service or the TCP SYN packet, until the socket is closed.

When clients make requests via the rate.d protocol, they incur an added delay of one round trip time plus any processing overhead for job assignment. The round trip delay results from the initial rate.d request, which must be sent prior to initiating a TCP connection. In our experimental environment, the round trip time was negligible; in practice, round trip times may be as high as a few hundred milliseconds depending on network conditions.

At low request rates, when the servers can handle all requests, all job assignment systems deliver approximately the same latency. Table 1 shows the average latency observed when the total request rate was 200 RPS or less.

Assignment mechanism	Average latency (in seconds)
Rate.d	1.8
LocalDirector	1.9
Round Robin	2.0

Table 1. Average latency for five-server cluster

At higher request rates, we measured both the latency and the number of requests that timed out. A timed out request is one that did not complete during the test run. The latency reported for these tests is the average for all connections that were completed. Figure 6 and Figure 7 show the latency and timeout rate for 5 servers and Workload A.

Rate.d is approximately twice as fast as round robin assignment and three times as fast as LocalDirector at high loads, although latency is very variable at high loads. All three mechanisms time out approximately the same number of requests, although LocalDirector performs somewhat better for a small fraction of the test runs. The improved latency provided by rate.d is a result of the good overload behavior described in the previous section. By discarding jobs that it will not be able to handle, the cluster makes faster progress on those jobs it does accept.

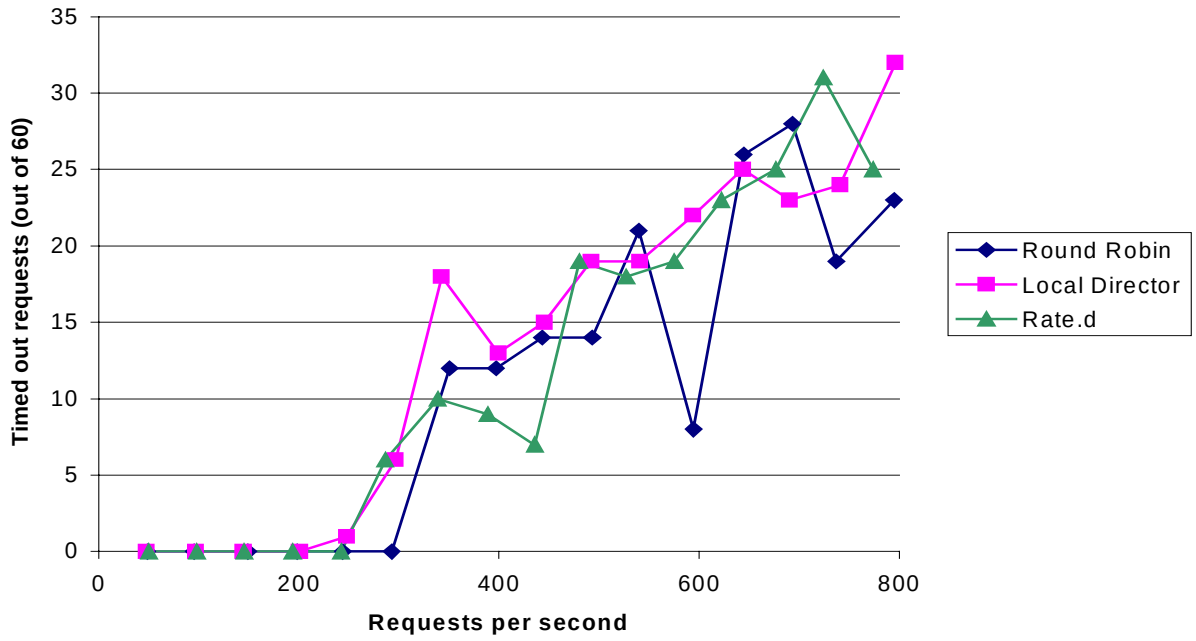

Figure 7. Timed out requests on a 5-server cluster

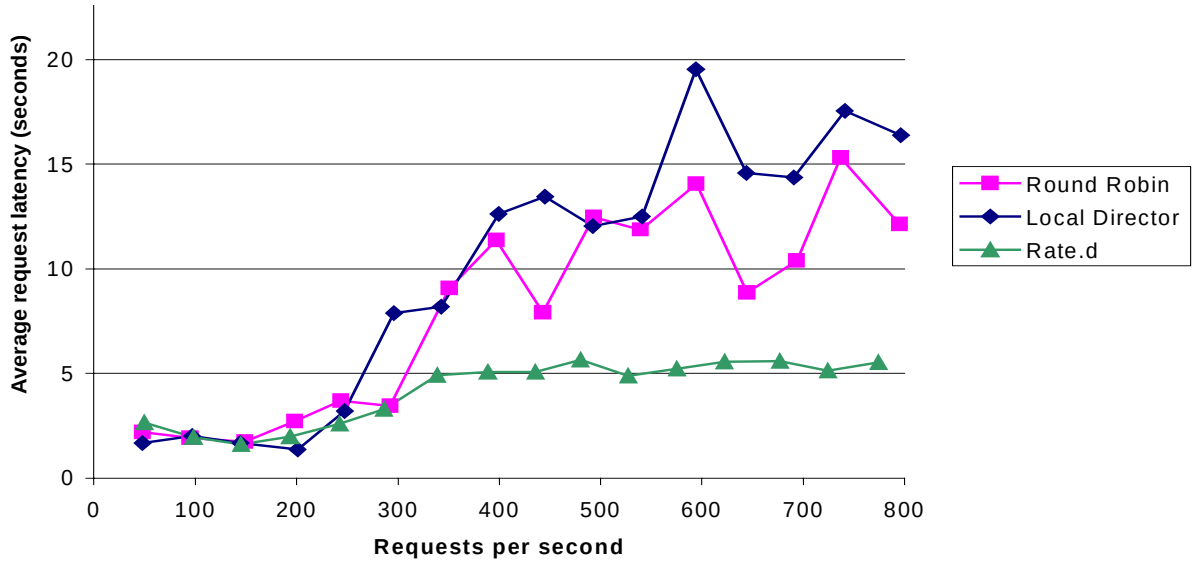

Figure 6. Latency on a 5-server cluster

Performance on different cluster sizes

Rate.d offers approximately the same performance on clusters with one to five machines. Figure 8 shows the throughput of clusters with one to five machines for Workload A. The performance curve for each cluster has the same basic shape. The cluster offers nearly optimal performance at low rates, gradually reaches its peak performance, and then levels off at the peak rate. (The one outlier in the 4-server curve is the result of a server being swamped as described in "Throughput," above.)

Detailed Analysis of Rate.d Behavior

This section describes rate.d behavior in more detail for Workload A on a 5-server cluster. Specifically, we discuss duplicate responses to the initial Request for Service, dropped requests, and CPU time consumed by rate.d. Figure 9 shows rate.d behavior in terms of packets produced and lost at request rates ranging from 50 to 800 RPS.

Duplicate responses

The rate.d algorithm can result in several responses being sent in response to a single Request for Service. Although it is difficult to quantify the effect these duplicate responses have on performance, the analysis does help explain how the rate.d algorithm behaves in operation.

In theory, duplicate responses can occur either because the servers disagree on the results of the initial lottery or because some problem causes a second server to send a response because it mistakenly thinks the lottery winner did not. In our implementation, there are three immediate sources of duplicates:

1. Timing issues in the rate.d implementation, which cause internal updates to be delayed at either sender or receiver.
2. Lost internal update packets.
3. Disagreement among servers about the current system state, which results in different lottery outcomes at different servers.

In practice, the first factor dominates. For example, it is possible for rate.d to fall behind on processing internal broadcasts messages, so that it runs a new lottery for a request even though it has a commit message queued for processing. These timing issues are detailed in "Implementation Issues," below. We did not measure the effects of disagreement on system state.

Figure 9 shows that the duplicate response rate increases from 30 percent at 50 RPS to almost 60 percent at 250 RPS; in other words, at the peak rate six out of every 10 requests results in two or more responses. The duplicate rate drops off as the request rate increases beyond 250 RPS.

The duplicate rate is consistent with our expectations, given the following intuition: The probability of receiving a duplicate request is largely a function of the number of servers that see the request. At low request rates, every server sees every request and the duplicate rate increases linearly. At higher request rates (beyond the peak rate), each server

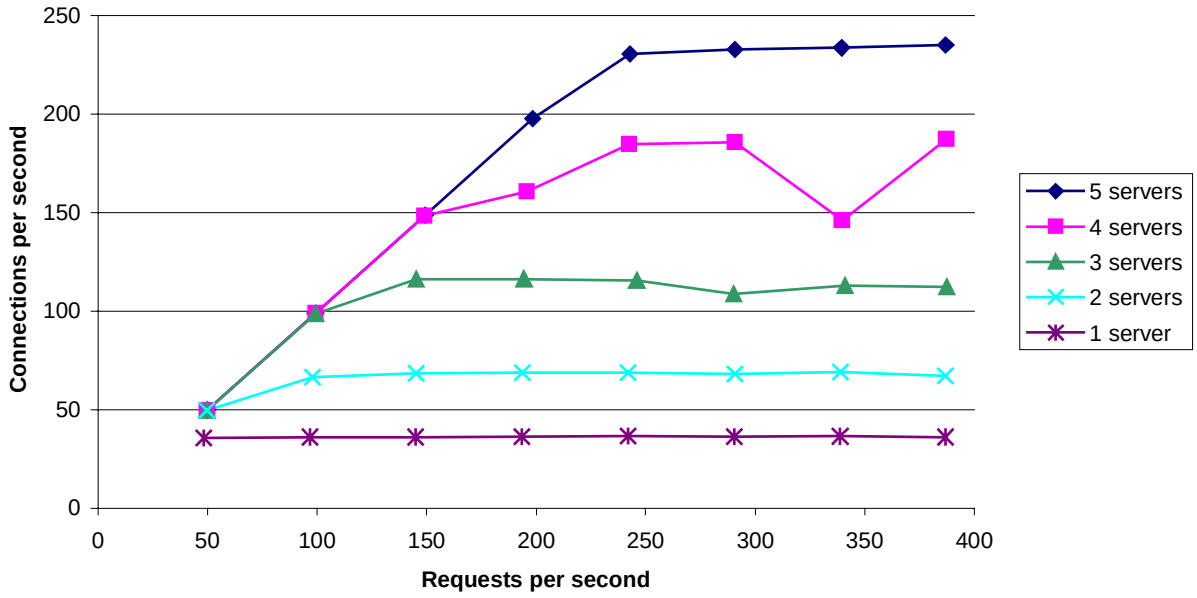

Figure 8. Rate.d performance for different cluster sizes

sees a different subset of the request stream, because it drops some requests. Thus, the probability that any particular request receives a duplicate response is lower because, in general, it was seen by only part of the cluster.

While we have not measured the actual effect of duplicate responses, we briefly discuss two possible effects.

Duplicates generate unnecessary traffic on the cluster's internal network and on the client's network connection. The effect on client connections is negligible, because the number of duplicates for any single request is small. On the cluster network, the actual application payload probably generates much more traffic than rate.d responses.

When rate.d sends a duplicate response, it may be wasting CPU time. For cases where two servers respond to the initial request, the extra work is minor—an extra UDP message sent. In the other case, the rate.d that sends a duplicate must run a second lottery for the job and then add the request ID to its next internal broadcast. Checking pended jobs is a bottleneck in the current implementation, so checking fewer pended jobs would decrease the amount of work rate.d does.

We expect that these effects can be addressed by future work.

CPU time used by rate.d and lost responses

CPU time increases linearly with the number of requests. When the server reaches its capacity (in jobs per second), the application server and rate.d start to compete for CPU time. When there is contention, rate.d does not receive enough CPU time to keep up with incoming requests. As a result, it starts dropping incoming requests. This can be understood as a self-limiting feedback loop. Figure 10 shows the amount of CPU time used by rate.d running on 4-CPU Pentium Pro servers and 2-CPU UltraSparc servers.

The CPU contention at high requests rates leads to a kind of equilibrium. Because rate.d drops packets, it accepts fewer jobs. If the amount of CPU time decreases, it drops fewer jobs and takes more jobs. The reverse is also true.

When rate.d does not get enough CPU time, it drops packets. As the request rate goes higher, the number of jobs handled remains constant. All the extra requests turn into drops. At somewhat less-than-capacity request rates, we see a handful of dropped requests, less than 10.

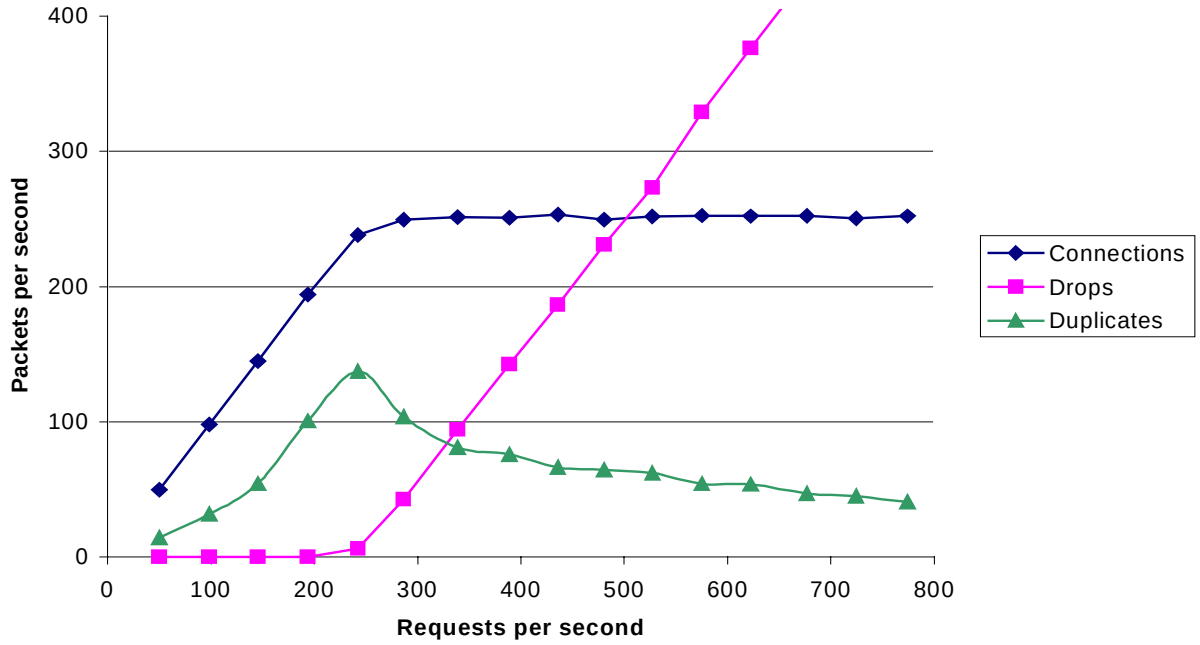

Figure 9. Rate.d response packets

Configuration Issues

The load-balancing behavior of the cluster depends on the configured maximum capacity of each machine and on the algorithm used to derive the current "availability" from this maximum capacity. Neither issue has been carefully studied with the current implementation and would benefit from further research.

The current approach of computing current availability as a fraction of the maximum capacity is problematic. In practice the capacity of a machine depends on characteristics of the service being offered, so a machine's capacity rating must be re-configured each time the workload changes. As a result, there is no effective means to configure a capacity rating that will be good for a workload that includes multiple services. It is also difficult to choose a good capacity rating for a server. In our tests, we have assigned capacity by first running a workload on a cluster equipped with LocalDirector and then assigning capacity based on the weights assigned by LocalDirector.

Rate.d computes the current availability from the maximum capacity based on the number of runnable or blocked processes on a server. Once per second, the operating system reports the number of processes that are runnable or that are blocked waiting for some resources (such as I/O). The availability is defined as the maximum capacity divided by the number of runnable or blocked processes. This algorithm has several unfortunate consequences.

In particular, the availability of a machine approaches zero quickly. As the number of processes increases, the relative difference between machines of differing capacities becomes negligible. One must also be ensure, when configuring the system, that the maximum capacity is much larger than the expected number of processes the system will handle; if the numbers are of similar magnitude, the availability could drop to zero. Finally, the scaling effects of the algorithm are poor; a machine with two processes will have an availability rating 50 percent higher than the same machine with three active processes.

Implementation Issues

One implementation problem has caused significant performance problems and is briefly described here. The rate.d algorithm relies on fine-grained timing that is supported poorly by the current implementation. This timing issue is not intrinsic to the rate.d approach, however. A different implementation could avoid the timing issue.

For example, each rate.d daemon must keep track of jobs that should be taken by another server and later check that the other server in fact took them. (A job that is supposed to be handled by another server is called a "pended job" because rate.d might need to respond to it if the other server doesn't.) The check must occur long enough after the initial

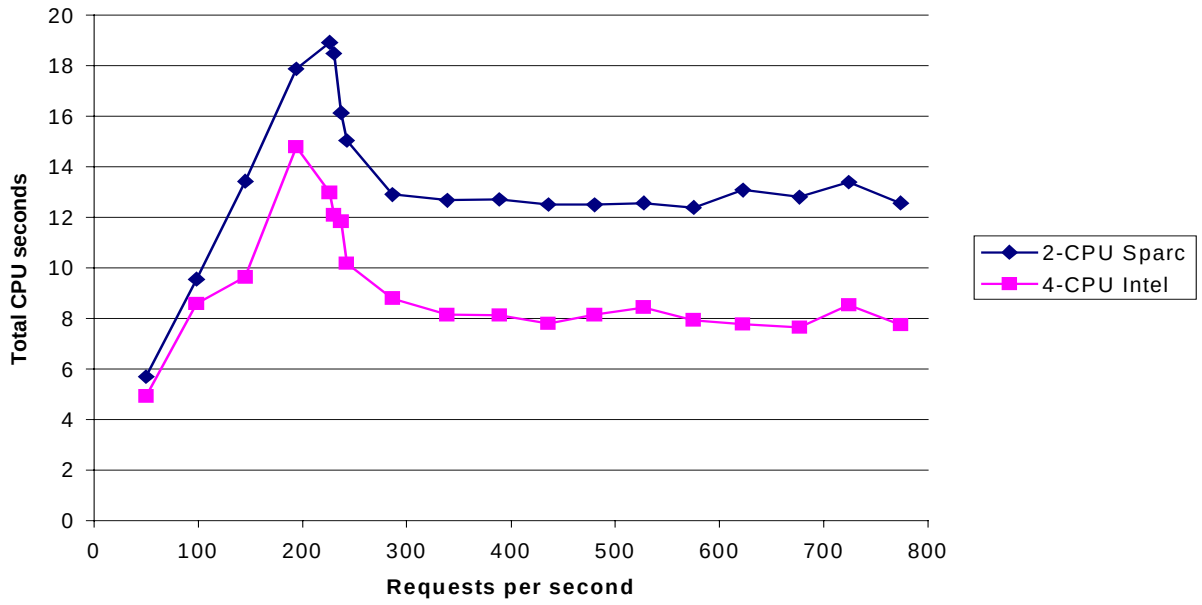

Figure 10. Rate.d CPU seconds for 3 minute run

request that there is time to receive and process an internal broadcast message from the server that took the job. The check must not be delayed too long, however, or client response time suffers.

The implementation uses an event queue to schedule actions like checking pended jobs or sending internal broadcasts. The rate.d main loop checks sockets for incoming messages (either client requests or internal broadcasts) and then it runs any actions in the event queue. If any of these actions take a long time to run, rate.d will fall behind on processing incoming messages. In extreme cases, the operating system will drop incoming packets and rate.d will never see them.

Unfortunately, rate.d checks the status of all pended jobs once per second using the event loop. Under heavy load, the number of pended jobs is large (several hundred), and it can take several hundred milliseconds of CPU time to check all of them. Because rate.d is competing with other processes for CPU time, it can take more than a second of real time to check pended jobs. These delays increase request latency and causes excessive losses.

Results for Workload B

Workload B overloads the networking hardware in our test environment. The cluster is only partly loaded when the network's capacity is reached and performance levels off. A Rate.d performance lags behind LocalDirector and round robin for this workload. The peak rate.d performance for any test run, with 3 or 5 servers active, is between 450 and 500 RPS. Round robin and LocalDirector tests both exceed 500 RPS. This is shown in Figure 11.

It was not possible to make accurate measurements of latency for Workload B, because the network saturated before the servers. The latency that was measured suggests that the three systems deliver similar latency for workloads A and B. However, at high loads for round robin and LocalDirector, packet loss at the router presumably skews the measurements.

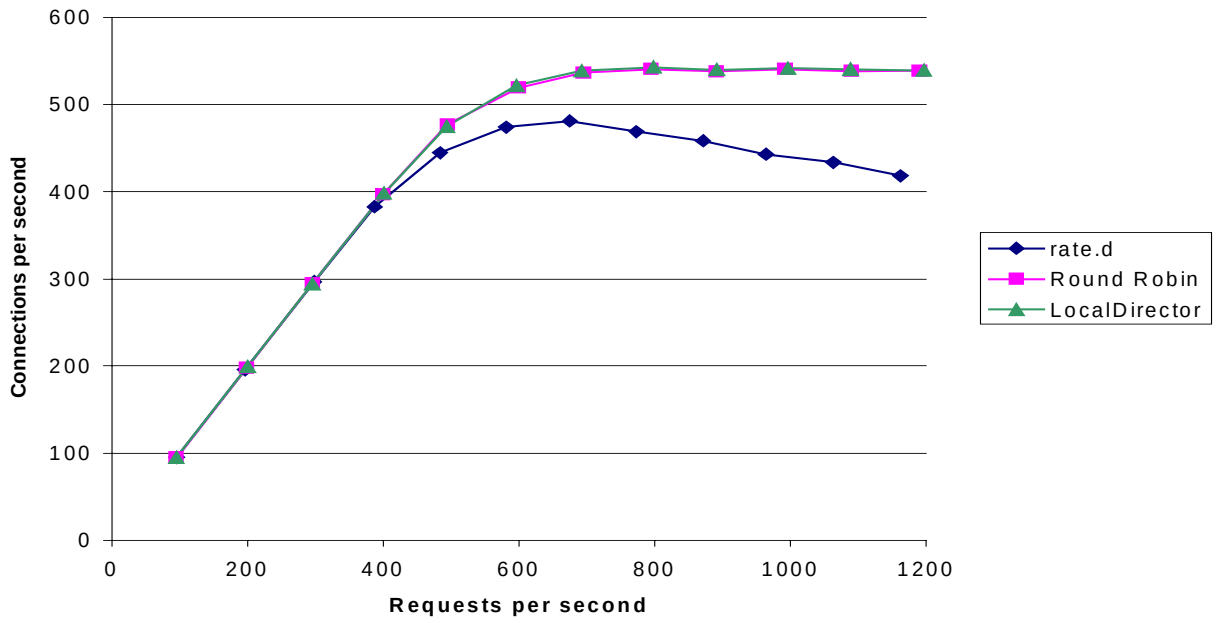

Figure 11. Throughput comparison for workload without CGI requests

Fault Tolerance

We made one set of measurements to compare how rate.d and LocalDirector behave when one server fails. (Round robin selection was ignored because it has obviously poor fault tolerance; clients do not know when a server is down and will continue to attempt connections to it.) During a six-minute test run, one of the Dell servers was disconnected from the BayStack switch for exactly one minute. The aggregate client request rate was 200 requests per second,

which is low enough that the four remaining servers would not be overloaded during the one-minute outage. The goal of this test was to measure the effect of a system failure during normal, non-overload operation.

Both systems perform well when one server fails. Rate.d successfully handled 98.5 percent of all incoming requests. LocalDirector successfully handled 97.5 percent of all incoming requests. (All measurements are based on the average of two test runs.) The requests that were not handled by the system timed out during the initial connection establishment phase. Requests that received no response within 1 second timed out; note that the latency between client and server is negligible (about 1 ms), so the timeouts result from delays processing the request inside the server cluster.

There was no measurable difference between the average request latency for normal tests and for the failure tests.

Test runs did show one unusual effect in rate.d's behavior after a server recovers from a failure. In the tests, when the "failed" server was reconnected to the network, it received a very high number of jobs during its first second of operation. The number of jobs committed to during that second was more than 3 times the average number of jobs per second during the rest of the test run (excluding the time when the server was disconnected). One test showed two such surges in the first 5 seconds of operation. However, the high commit rate did not appear to have significant effects on overall performance.

Future Work

A number of improvements to the system need to be made before it can be considered to be production quality: more and better management tools (including automated remote installation of new services), a faster and more robust implementation of rate.d, and so on.

More interestingly, the project suggests several new research directions.

- Currently, rate.d requires the client software to be modified, or else it requires the use of a special rate.d proxy service, which takes away most of the advantages of the protocol (reliability and some scalability). A possible approach would be to move the job assignment decision to the TCP connection establishment.
- The current rate.d algorithm scales poorly as the cluster size grows, because each request needs to be seen by each server. We need to re-think the job assignment mechanism so that not all servers have to process each incoming request.
- Proper functioning of rate.d depends on adequate ways to measure the load experienced by a server (and to represent it as a single number or a small vector). The current measurement is very ad-hoc; a better approach should be developed.
- The current system assumes that all servers in the cluster are co-located. An interesting topic for new research would be to drop this assumption and design a system for load balancing between servers located at different places on a wide area network.

Conclusion

We have described the design and a proof-of-concept implementation of the Application Gateway System, a fully distributed server system that runs on a cluster of heterogeneous workstations. The system was evaluated using a five-server cluster running an HTTP application workload; the evaluation was based on comparisons with the same cluster using a round-robin assignment and Cisco's LocalDirector. The system is transparent to end-users, automatically adapts to failures and reconfiguration, and offers near-optimal performance for modestly CPU-intensive workloads. It can be configured and managed using a secure management protocol.

We draw the following conclusions from our work:

- Rate.d demonstrates that it is possible to build a fully distributed job assignment system, although the current approach has scaling problems. Continued work is needed to develop a scalable solution.
- The AGS management system is a useful tool for configuring and managing the individual servers in a cluster.
- The LocalDirector product offers good performance for a wider range of workloads than rate.d, including those workloads where each job requires the server to do very little work, e.g. serving static files. LocalDirector is an effective solution.

Our evaluation of rate.d, LocalDirector, and round-robin assignment under overload conditions, i.e., when more requests arrive than the servers can handle, suggests that standard TCP implementations do not shed load gracefully. The use of UDP for rate.d requests led to better overload performance, presumably because UDP packets are handled differently by the operating system.

References

[ACAP]
Newman, C., Myers, J.G., "ACAP—Application Configuration Access Protocol." *IETF RFC 2244*, Nov 1997.

[Rate.d]
Drake, Fred L. Jr., Masse, Roger E.; Warsaw, Barry. "Rate.d Distributed Load Balancing System." Corporation for National Research Initiatives, 1995-1997.

[RFC 1631]
K. Egevang, P. Francis. "The IP Network Address Translator (NAT)." *IETF RFC 1631*, May 1994.

[SClient]
Banga, G. and Druschel, P. "Measuring the Capacity of a Web Server." *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey*. December 1997.