

Distributed packet routing using adaptive hash functions

Jared Jennings

7 May 2003

Abstract

As traffic on the World Wide Web increases, more power is needed to serve web pages, especially with the advent of sites like Slashdot (and, with them, adjectives like 'slashdotted'). But large, powerful servers are expensive and proprietary. An alternative is to split up the task of serving web pages between multiple, smaller servers. Various solutions have been proposed, such as round-robin DNS, which redirects requests by answering DNS requests for a site with one of several IP addresses; redirection, which depends on servers whose sole purpose is to hand out HTTP redirects to the real web servers; and fancy packet routing, i.e. placing a router at the head of the network which routes incoming connections to different servers depending on the source or other parameters of the request. Another solution is discussed and expanded upon: distributed packet routing. In this strategy, not only is the task of serving web pages split between the servers, but so is the task of routing packets. We discuss a better way to distribute incoming requests using an adaptive non-uniform hashing strategy, which allows for changing capacities due to load on the servers, and even partial outage of the server cluster, while guaranteeing graceful degradation. Request packets arrive at any of the servers at a site, thanks to round-robin DNS, and each server can forward requests to other servers to make the load distribution more even than round-robin DNS can, allowing for smoother scalability and less unpredictable response times as overall load goes up. The SIEVE strategy used for choosing servers guarantees even distribution and allows for both new servers entering the system, and server capacities changing due to load or outage.

1 Background

1.1 Rising web server traffic

The number of requests to most web sites is increasing as the number of users on the Internet increases, and this growth requires commensurate growth in the capacities of the servers that run the sites. The growth function is of exponential order, not linear, so a great degree of latent capacity is needed. Organizations that run small sites don't want to buy large servers to begin with (and buying ahead of the curve in the IT market is presently not deemed to be a good investment), but they also don't want much downtime as the inevitable need to switch to larger servers comes. Single-system-image servers are limited in size (presently the maximum is somewhere around 128 processors) and prohibitive in cost, because of the issues involved in, for example, arbitrating access to one memory image from 128 processors, or maintaining cache coherency between 128 level-2 RAM caches.

1.2 Distributed web systems

It is possible to craft a web server from multiple smaller servers, such that each server takes some of the incoming traffic and responds to it; the primary problem with this scheme is how to split the incoming requests among the servers for optimum performance and scalability. Many ways have been suggested for doing this, and the various solutions are well classified in [CAR02]. According to the nomenclature therein, the solution proposed here is a *distributed Web system*.

Most systems so formed are homogeneous, i.e., each server in the system is configured identically; and they are easiest to set up and administrate if all servers are present at the time of startup. But it is useful if a distributed web server scheme allows for the use of servers of different capacities. This allows the maximum possible amount of flexibility as the traffic, and the system, grows. It is even more useful if a strategy allows for adding servers without taking the system down. In this case the hardware needed for the system can be brought online incrementally as needed. If a sudden leap in capacity is needed, a larger server can be added to the system on the fly with no system downtime.

2 Related work

2.1 Round-robin DNS

Round-robin DNS [KAT94] is the seminal idea and paper on the subject of distributed web servers. The idea is that the DNS server hands out different IP addresses to different clients as corresponding to the name (for example) `www.somecompany.com`. So client A looks up `www.somecompany.com` and gets `11.22.33.40` from the DNS server `dns.somecompany.com`, client B making the same lookup gets `11.22.33.41`, and so on depending on the number of servers at the site.

This scheme does in fact distribute the load of processing requests between servers, but not as evenly as was originally hoped. The problem is that it is not a reliable proposition that a DNS lookup will be fast, so name-address mappings are cached at many levels: on the client computer itself, on the DNS server of the client computer, and potentially all the way on up the hierarchy to the root servers. So if someone from AOL visits `www.somecompany.com`, AOL's proxy servers cache the mapping from `www.somecompany.com` to `11.22.33.43`, and anyone visiting `www.somecompany.com` from AOL hits that webserver and no other until the DNS time-to-live for the mapping has expired (typically 1-3 days).

This problem can be ameliorated by shortening the time-to-live for the DNS record for `www.somecompany.com` on `dns.somecompany.com`, but this results in many more DNS requests coming in, and suddenly the need for distributing web server load has turned into a need for distributing DNS load, which is a separate and somewhat more difficult problem. So round-robin DNS can serve as a starting point for distribution of web server load, but not as a complete strategy.

2.2 Hierarchical redirection

Hierarchical redirection [MOU97] employs a number of web servers and a number of redirection servers. The redirection server for a particular request is chosen by round-robin DNS. The redirection servers know the loads of the web-servers (which can be defined in any pertinent way, such as number of running processes, CPU usage, network bandwidth usage, etc.), and their sole purpose is to hand out HTTP redirects pointing to the web servers themselves. This allows the load to be balanced with much finer granularity. So, for example, a browser attempting to fetch `http://www.somecompany.com/` would receive an HTTP redirect to `www3.somecompany.com`, whereupon it would connect to that server in order to retrieve the actual web page.

Files are shared between web servers with a network file system, and cached on each web server. If space is lacking, each server is not required to have a complete copy of the website, because, in addition to knowing which web servers have the lowest loads, the redirection servers also know which web servers have which files and directories on their local storage, and can redirect requests for those files to such servers. The list of storage locations of files and directories must be kept small for quick lookups, so usually files should be placed on the local storage of the web servers a directory at a time. Then the redirection servers determine to which web server to redirect by looking at whether the file is inside each of the directories in the list.

There are two problems with the redirection-server strategy. The biggest is bookmarks: clients can make bookmarks for pages they've visited. Suppose our client, which was redirected to `www3.somecompany.com`, makes a bookmark for the site. Every time the bookmark is opened, a request to `www3.somecompany.com` will result, regardless of the current loads on the web servers in the cluster, because the bookmark circumvents the redirection scheme. Further, suppose the bookmark was for a file inside the website, like `www3.somecompany.com/files/pages/foo/bar.html`. It is possible for files to be moved around between the webservers such that `www3` no longer has a local copy of `/files/pages/foo/bar.html`. The server must then fetch the file from another server in the system via the network.

So the redirection scheme, while much more promising than round-robin DNS, creates some problems of its own, the solutions to which can become complex, and the scheme is too easy to thwart with bookmarks.

2.3 Packet routing

Instead of seeking to direct the client browser to a particular server using HTTP or DNS, packet-routing solutions go to a lower level and translate TCP/IP connections, all destined to the same IP address, to different servers in the system (see [AND96]). Suppose, for example, that a client connects to `www.somecompany.com`. The server actually accessed only routes the connection request and all further packets from this client during the connection to one of the actual web servers (and likewise with all packets coming back from the server). This equalizes server load much better than round-robin DNS, but for many requests, a high load is placed on the router, shifting the load from the servers to the router.

2.4 Distributed packet routing

The answer, of course, is to distribute the load of packet routing as well as the load of web service [AVE00]. In this scheme, each web server knows the loads of all the web servers in the system (again, “load” can be defined in any pertinent way). If it receives a request and its load is above some threshold and there is another server with less load, it wraps the TCP/IP packet requesting the connection in another IP packet bound for one of the other servers. If a server receives an IP packet within an IP packet, it unwraps it and processes the connection and ensuing request. This offers more fine-grained control of where requests go than round-robin DNS like the HTTP redirection scheme, and it cannot be so easily circumvented by normal user behavior. But the authors only explored a small number of algorithms for deciding which requests are forwarded and to which servers, the implementation was made for an old version of the Linux kernel, and most importantly, the code they wrote was not available.

3 The new implementation

While the distributed packet routing implementation of Aversa and Bestavros [AVE00] involved modification of the routing code in the Linux 2.2 kernel, the new implementation builds on the more current 2.4 kernel, and involves less new code in the kernel, improving portability and compatibility with future kernels. Instead of IP tunnelling, the new implementation only involves network address translation (NAT), the rewriting of source and destination addresses in packets. This eliminates the overhead of encapsulation and decapsulation involved in tunnelling.

3.1 Routing

Our implementation is based on the combination of *netfilter* and *iptables*, which together form the packet filter included with Linux 2.4. *netfilter* is a more flexible framework than its predecessors for implementing anything inside the kernel which inspects or modifies packets.

netfilter simply consists of five hook points, numbered 1 through 5 in the illustration, and the code to hook and unhook custom functions from these hook points. *iptables*, in turn, is a flexible packet filtering, NAT and packet mangling package which hooks itself into *netfilter*.

In order to use *netfilter* and *iptables* to change the routing of packets, first each host in the system is assigned a private IP address (e.g., from the

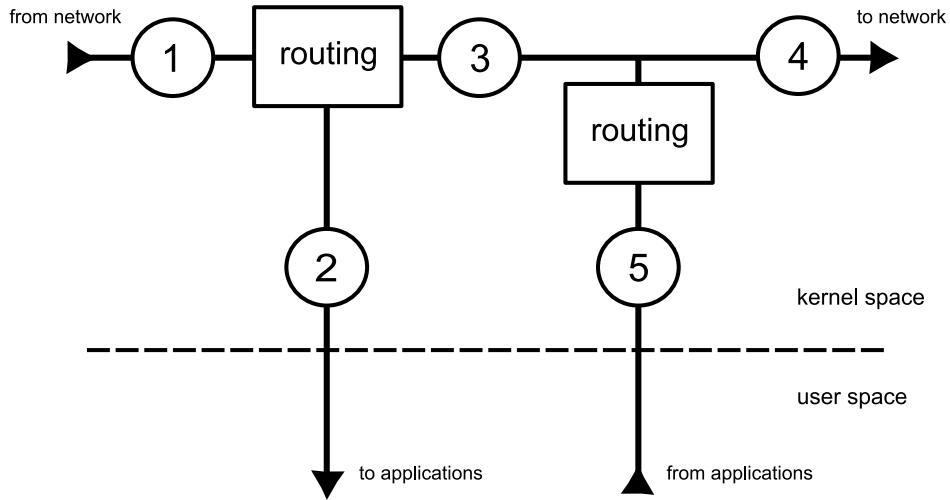


Figure 1: The netfilter hook points [RUS02].

range 10.x.x.x) which is on a network with the other hosts. Packets coming in whose destination is this private IP address are let through unchanged.

Connections coming in to port 80 (the default HTTP port) of the public IP address of each host are “marked,” that is, a variable inside the kernel which is associated with each packet, but not inside the packet, is set to a certain value. This value will determine where the connection goes. If the connection is to be rerouted and not served locally, the destination addresses of the packets are rewritten from the public IP address of the host to which the connection was made to the private IP address of the machine to which they are to be routed, based on the value of the mark. This happens at the prerouting hook point, point 1 in the illustration. The routing code then determines based on the destination address that the packets are to be forwarded, not given to applications on the routing host. Outgoing packets which are destined to a private IP address have their source rewritten to the private IP address, so that the connection now appears not to be from a client outside to the host doing the routing, but instead from the routing host to the destination host. This is done at the postrouting hook point, point 4 in the illustration.

The NAT code in *iptables* makes tables of which connections had their packets rewritten and to what values, and rewrites return packets in the opposite way the original packets were rewritten. So reply packets coming from the destination host return to the routed host (the apparent source

of the connection), where their destination addresses are rewritten from the private IP address of the routing host to the IP address of the client (at hook point 1 in the illustration), and their source addresses are rewritten from the private IP address of the destination host to the public IP address of the routing host (at hook point 4 in the illustration). In the end, the client sees the routing host as the source of the reply, when it did not actually process the connection, but only forwarded it.

For example, suppose that Foo Company wants to use distributed packet routing to serve its website, `foocompany.com`. FooCo has an internal network with IP addresses of the form `10.x.x.x`. Public IP addresses are of the form `0.x.x.x`. A client `C` wants to access `http://www.foocompany.com/`. `C`'s IP address is `0.4.3.5`. FooCo has five web servers, `S`, `T`, `U`, `V` and `W`, with public IP addresses `0.2.2.1` through `0.2.2.5`, and private IP addresses `10.0.0.1` through `10.0.0.5`. FooCo also has a DNS server `D`.

`C` first looks up the IP address of `www.foocompany.com` using the DNS server, `D` (dotted line in illustration). `D` is using round-robin DNS, so `C` may get any address in the range of `0.2.2.1-0.2.2.5`. (Recall that this in itself does not yield even distribution of the load.) Suppose `D` answers `0.2.2.2`, the address of `T`. `C` then makes an HTTP connection to `T` (heavy line). The packet filter rules loaded into `T`'s kernel dictate that the packets of this connection, which is to the HTTP port 80, must be marked. The correct mark is calculated (see next section), and applied to the packets of the connection. Suppose the mark is 5. The NAT rules in `T`'s kernel specify that packets marked 5 should be routed to server `W`. The destinations of the packets are rewritten from `0.2.2.2` to `10.0.0.5`. The packet routing code in `T`'s kernel routes the packets towards `W`. Just before going out, the post-routing NAT rules rewrite the packets' source address to `T`'s internal address, `10.0.0.2`.

`W` gets a connection from `T` (`10.0.0.2`) to itself (`10.0.0.5`). It serves the connection normally. When the packets of the reply hit `T`, the NAT code undoes its former rewriting by setting the destination address (which was `10.0.0.2`) to `C`'s address (`0.4.3.5`). The kernel's routing code determines that the packets are headed out to the Internet. Before they go, the NAT code sets their source addresses, which were `10.0.0.5` (`W`'s internal address), to `0.2.2.2`, `T`'s public address. `C` only sees a connection from `C` to `T`, and `W` only sees a connection from `T` to `W`.

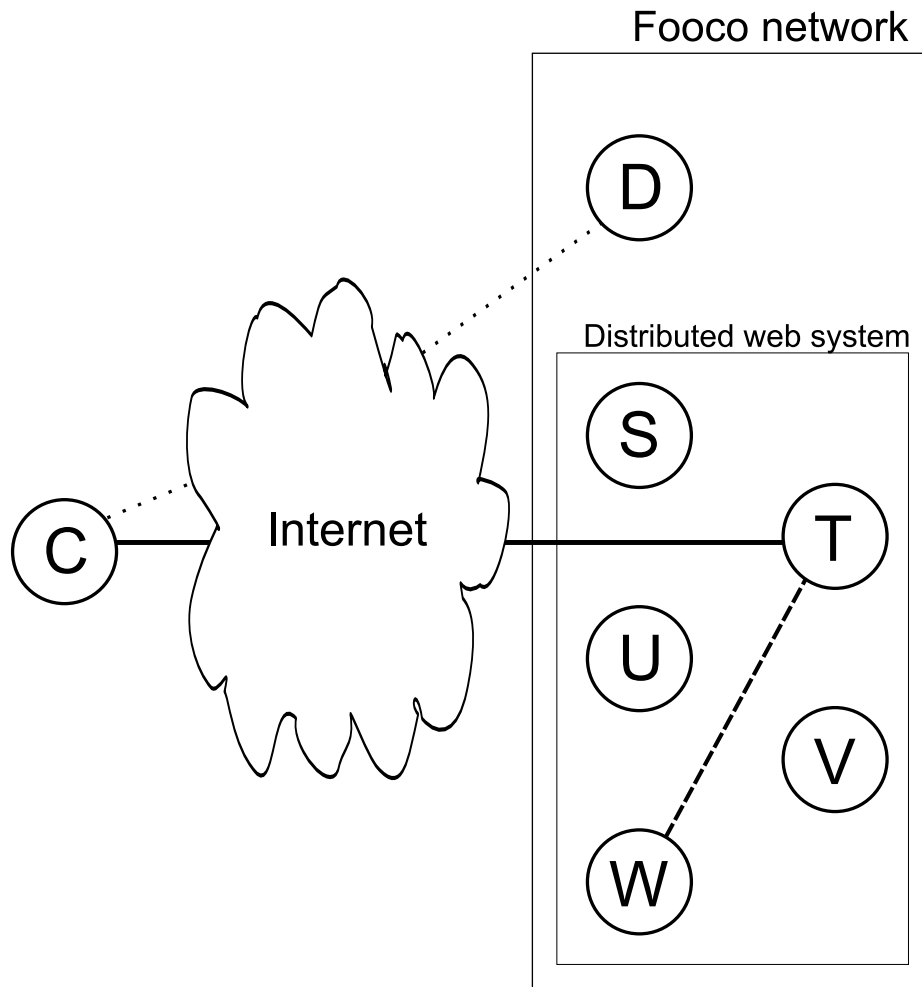


Figure 2: An example setup.

3.2 Marking strategy

The packet routing scheme in the previous section will perform the task of distributed packet routing suitably, given a heuristic for marking the packets. They can be left unmarked, in which case each host will serve the connections coming to it, irrespective of its load or capacity; this is the null functionality. They can be marked randomly; this might distribute the load better, but it also might not. A better strategy is needed.

Hashing functions are efficiently computed, have predictable results and can guarantee even distribution. Adaptive hashing functions can make these guarantees under changing numbers of hash bins. Nonuniform hashing functions can hash into bins of unequal size. The combination of these three characteristics results in a function which, given the capacities of each of the hosts, can mark packets in such a way as to evenly distribute connections between servers despite changing and unequal capacities. We use here a variant of the SIEVE strategy described in [BRI02]. For greater succinctness and generality when proving theorems they present, Brinkmann, Salzwedel and Scheideler assume the capacity of the entire system to be 1, and that the individual capacities are fractions. But in kernel code, floating point math is not used, for reasons of portability (not all processors have floating-point units), speed, and to keep the kernel from overwriting any user data in the FPU.

We keep a table of hosts in the system and their current capacities on each host. The capacities are dynamically updated. Given n hosts (“bins” in more general hashing parlance), we say that the capacity of host i is d_i , and $\sum d_i = d$, the total capacity. We keep n' ranges or intervals, where $n' = 2^{\lceil \log n \rceil + 1}$ (e.g., for $n = 5$, $n' = 16$). Each host is allowed to own any number of full ranges and at most one partial range. The total share x_i of the ranges that any host i owns is

$$x_i = \frac{d_i n'}{2d}$$

The number of full ranges a host owns is $\lfloor x_i \rfloor$ and the number of partial ranges is 0 if $x_i = \lfloor x_i \rfloor$. Asymptotically, given these assignments, half the ranges are assigned and half unassigned. The unassigned ranges belong to the “fallback host,” the host with the largest capacity.

We keep a list of L random functions of the form $f(x) = ax + b \pmod{\text{MAX_ULONG}}$ where MAX_ULONGLONG is the maximum value that can be stored in an unsigned long integer variable, and $L = \log n' + f$ where $f \geq 0$ is a number chosen to ensure even distribution. The following algorithm yields the mark for a connection:

```

function sieve_hash(x, hosts, ranges, functions)
  for count = 0 to L
    h = owner of ranges[functions[count](x)]
    if h <> NULL then return index of h in hosts
  return index of fallback host

```

Figure 3: The SIEVE hashing algorithm.

Theorems proposed and proven in [BRI02] include guarantees about the maximum amount of storage required, the maximum “unevenness” of requests across hosts, the efficiency of the algorithm. These assure that within a small number dependent on n and f as defined above, each host in the system will get a number of requests routed to it over time which is commensurate to its relative capacity within the system. If that capacity is 0, no other host in the system will route connections to that host, thus avoiding timed-out connections and keeping response times low.

3.3 Marking implementation

A small userspace daemon runs on each host in the system, checking its load every two seconds. Load here is defined as number of TCP connections open on a host; many other measures of load are valid and can be used. The current capacity is calculated as the base capacity of the host minus a function of the load, so heavily loaded machines have less capacity to serve further requests. The daemon then sends the capacity of the host to every other host in the system, over the network. The daemon also receives all of this data from the other hosts in the system.

A pseudo-device is created by a loadable kernel module `dpr.o`. This device is visible as `/dev/misc/dpr`, into which the daemon writes the received load data. The kernel module keeps track of the load data written to it, and makes it available to the packet marking code, which uses the SIEVE hash function to mark packets corresponding to requests as they come in. Whenever the capacity of a host has changed, then, within two seconds the host sends its changed capacity to itself and the other hosts. Every host recalculates the number of ranges belonging to the host with changed capacity, and updates its list of ranges to suit the new number of ranges owned by the host with changed capacity, in a concurrency-safe way. As requests come in, the SIEVE hash function uses the list of ranges as it stands.

4 Test results

5 The future

Up to now, the discussion has centered around serving files to clients, on the basic assumption that none of the files involve any computation at the time of the request or any resources specific to the host from which they came. In actuality, most web pages shown today do involve computation on the fly; while some files (such as pictures) are still usually static, most of the HTML involved is dynamic.

So the problem involves not only spreading the load involved in sending files over the network, but also spreading the load of computing dynamic pages. Matters are complicated by the fact that, usually, there is some state stored on the server related to the state of the “session,” such as whether a user has authenticated, form values already filled in, or open files or database connections.

If this load distribution strategy were to be extended for use in dynamic websites, it would be more useful to speak of “sessions” than “requests,” and given the use of the adaptive placement scheme mentioned above, once a session is assigned to a server, it would be quick and easy for any server in the system to locate a session. The SIEVE strategy includes consideration for moving requests between hosts when capacities change, to keep things evenly distributed. If this were added to a scheme for migrating sessions between servers in the system, an excellent and fault-tolerant load-balancing scheme for session-based dynamic web service could be established.

References

- [AND96] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter, an application of fast packet interposing. <http://www.cs.berkeley.edu/~eanders/projects/magicrouter/>, 1996.
- [AVE00] Aversa, L. and Bestavros, A. Load balancing a cluster of web servers using distributed packet rewriting. In *IEEE International Performance, Computing and Communication Conference*, 2000.
- [BRI02] Brinkmann, A., Salzwedel, K. and Scheideler, C. Compact, adaptive placement schemes for non-uniform requirements. In *Proceed-*

ings of the 14th annual ACM Symposium on parallel algorithms and architectures, 2002.

- [CAR02] Cardellini, V. and Casalicchio, E. The state of the art in distributed web servers. In *ACM Computing Surveys*, Vol. 34, No. 2, pages 263-311, 2002.
- [KAT94] Katz, E. D. , Butler, M. and McGrath, R. A scalable HTTP server: the NCSA prototype. In *Computer Networks and ISDN Systems 27*, pages 155-164, 1994.
- [MOU97] Mourad, A. and Liu, H. Scalable web server architectures. In *IEEE Symposium on Computers and Communications*, pages 12-16, 1997.
- [RUS02] Russell, Paul. Netfilter architecture.
<http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-howto-3.html>, 2002.