

HashNAT: A Distributed Packet Rewriting System Using Adaptive Hash Functions

Hala ElAarag and Jared Jennings
Department of Mathematics and Computer Science
Stetson University
Deland, FL 32723
helaarag, jjenning@stetson.edu

Abstract

As traffic on the World Wide Web increases, more power is needed to serve web pages. 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 ways of doing this have been proposed, such as round-robin DNS, hierarchical redirection, and the use of load-balancing routers. In this paper we propose a distributed packet rewriting system. In this system, not only is the task of serving web pages split between the servers, but so is the task of routing packets. Incoming requests are initially distributed between servers using round-robin DNS; then the web servers themselves route connections as necessary to balance the load more equally. We use an adaptive non-uniform hashing function in routing connections. The use of this function guarantees even load distribution, and allows for both new servers entering the system and server capacities changing due to load or outage, with no downtime.

1. Introduction

The growth of the World Wide Web requires commensurate growth in the capacities of the servers that run its sites. Small sites do not want large servers to begin with, but need to avoid downtime as the inevitable need to switch to larger servers comes. Single-system-image servers are still limited in maximum size, but prohibitive in cost. Many solutions for distribution of the task of fulfilling page requests are classified and summarized in [4]. We propose another solution which, according to the nomenclature of the aforementioned paper, is a *distributed Web system*. The rest of this paper is organized as follows:

Section 2 serves as a summary of related work. Section 3 details our novel implementation of distributed packet rewriting using adaptive non-uniform hash functions, named HashNAT. In Section 4 we discuss the results of testing the HashNAT system. Section 5 presents the conclusion of the paper.

2. Related work

In round-robin DNS [5], the seminal idea and paper on the subject of distributed web servers, the DNS server hands out different IP addresses to different clients for the same name. The clients then connect to different servers to request the web page. So round-robin DNS distributes the load of processing requests between servers, but not evenly enough. Each server in the DNS hierarchy between the round-robin DNS server and the client caches the name-address mapping obtained, and answers future requests regarding that name with the cached address. This means that thousands of clients can hit the same web server while another web server goes nearly unused. This problem can be ameliorated by shortening the time-to-live for the DNS record, but this replaces too much web server load with too much DNS server load. So round-robin DNS can serve as a starting point for distribution of web server load, but not as a complete strategy.

Hierarchical redirection [6] puts redirection servers (chosen by round-robin DNS) between the client and the web servers. They determine which web server should get the request, possibly based on part of the URL or the loads of the web servers, and hand out HTTP redirects pointing to the documents on the web servers. This allows finer-grained load balancing than round-robin DNS. Also, the web servers need not all contain the same files, because the redirection servers can take the location of a requested file into account when redirecting requests. Problems with this strategy come when visitors make bookmarks for the web pages they've visited. The bookmarks point straight to a web server, circumventing the redirection mechanism. Worse, if the resource pointed to by the bookmark is removed from that web server's local storage, the server must fetch the file from another web server, over a very busy network. Hierarchical redirection, while more promising than round-robin DNS, can create some complex problems of its own.

Packet-routing solutions translate TCP/IP connections, all destined to the same IP address, to different servers in the

system (see [1]). Since they distribute requests at a lower level than HTTP redirection or round-robin DNS, they cannot be circumvented. They can equalize server load much better than round-robin DNS, but when the rate of requests grows, a high load is placed on the router, which can become a bottleneck.

Distributed packet rewriting (DPR) is a possible solution to distribute the load of packet routing as well as the load of web service [2]. In this scheme, each web server knows the loads on all the web servers in the system. 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 circumvented by normal user behavior.

3. The HashNAT system

In this section we present our distributed web system and show how it is different from previous techniques.

3.1. DPR using network address translation

We adopt the distributed packet rewriting idea by Aversa and Bestavros [2]. While their implementation involved modification of the routing code in the Linux 2.2 kernel, HashNAT builds with either the Linux 2.4 or 2.6 kernel, and builds outside the kernel source tree, improving portability to future Linux kernels and other operating systems. Instead of IP tunneling, we use network address translation (NAT), a widely used technique of rewriting the source and destination addresses of IP packets. This eliminates the overhead of encapsulation and decapsulation involved in tunneling.

In order to change the routing of packets using NAT, each server in the system is assigned a private IP address which is on a network with the other servers. Packets whose destination is this private IP address are let through unchanged. Incoming HTTP connections to the public IP address of each server 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 routed to another server and not served locally, the destination addresses of the packets are rewritten from the public IP address of the server 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. 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

server which is routing, but instead from the routing server to the destination server.

The connection-tracking code in *netfilter* [7] makes tables of which connections had their initial packets rewritten and to what values, and uses them to make the converse changes to return packets. So reply packets coming from the destination server return to the routing server (the apparent source of the connection), where their destination addresses are rewritten from the private IP address of the routing server to the IP address of the client, and their source addresses are rewritten from the private IP address of the destination server to the public IP address of the routing server. In the end, the client sees the routing server as the source of the reply, when it did not actually process the connection, but only forwarded it.

For example, suppose that XYZ University wants to use the HashNAT system to serve its website, `www.xyz.edu`. XYZ University 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.xyz.edu/`. *C*'s IP address is `0.4.3.5`. XYZ University 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`. It also has a DNS server *D*. (See Figure 1.)

C first looks up the IP address of `www.xyz.edu` using the DNS server *D* (dotted line in figure 1). *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

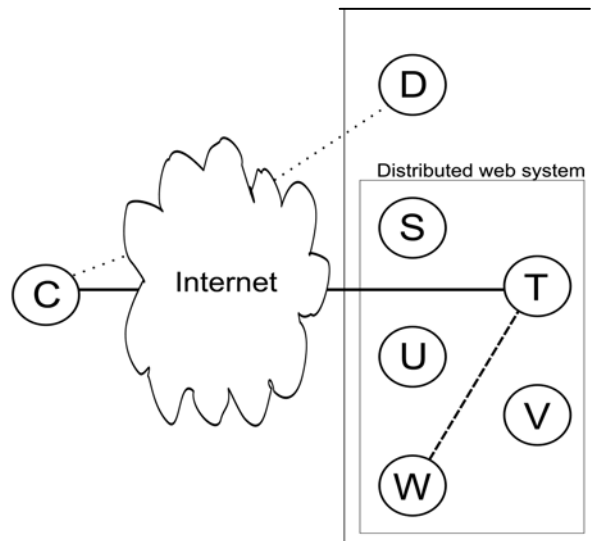


Figure 1: An example HashNAT setup, with client *C*, round-robin DNS server *D*, and web servers *S*, *T*, *U*, *V* and *W*.

80, must be marked. The correct mark is calculated as explained in section 3.2, 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 (dashed line). 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 the packets are sent to W, the post-routing NAT rules rewrite the source addresses of the packets to T's internal address, 10.0.0.2.

W sees a connection from T (10.0.0.2) to itself (10.0.0.5). It serves the connection normally (since the connection was made to its private IP address, not its public IP address). 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 based on their destination addresses 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.

3.2. Marking strategy

To calculate the mark to put on the packets, we use an adaptive non-uniform hashing function. The value of this function is quick to calculate, and its distribution can be made arbitrarily even. The argument to the function is derived from the IP address and TCP source port of the client, and the value of the function is an integer between 1 and the number of servers in the system, inclusive. Because it is an adaptive hashing function, the number of servers can change (e.g., servers can be added to the system with no downtime). Because it is a non-uniform hashing function, the odds of routing to one server over another can be adjusted as the loads of the servers change, and the servers need not have the same capacity for serving requests.

We use here a variant of the SIEVE strategy described in [3], which effectively turns one adaptive, non-uniform hashing problem into multiple adaptive uniform hashing problems. The original paper introduces and describes the SIEVE strategy in a mathematical context; but we use it as the cornerstone of our system because of the benefits just mentioned. A few changes to its implementation were necessary because of the environment in which we implement it. The discussions and proofs in the original paper fix the total capacity of the system at 1, and the individual capacities are real numbers between 0 and 1. But in the Linux kernel, floating point math is not used, so we count capacities of servers as integers between 0 and some large round number, and the entire capacity of the system as the (variable) sum of all such capacities.

We keep a table on each server of all servers in the system and their current capacities. The capacities are dynamically

```
function sieve_hash(x, servers, ranges, functions):
  for count = 0 to L:
    h = owner of ranges[functions[count](x)]
    if h <> NULL:
      stop and return index of h in servers
  return index of fallback server
```

Figure 2: The SIEVE hashing algorithm

updated as the loads change. Given n servers, we say that the capacity of each server i is d_i (where $0 < i \leq n$), and $\sum d_i = d$, the total capacity. We keep n' "ranges," where $n' = 2^{\lceil \log_2 n \rceil + 1}$ (e.g., for $n=5$, $n'=16$). Unlike the hashing bins which correspond to the servers, the ranges are of a constant size. Each server owns some number of them, and their use is the intermediary step that enables us to use preexisting, simple uniform hashing functions to make our non-uniform hashing function. The total share of the ranges that any server i owns is

$$x_i = \frac{n' d_i}{2d}. \quad (1)$$

The number of "full" ranges a server owns is $\lfloor x_i \rfloor$ and the number of partial ranges is 0 iff $x_i = \lfloor x_i \rfloor$. That is, a server owns one and only one partial range if its capacity does not give it an integral number of full ranges. Given this apportionment of ranges, roughly half the ranges are assigned to servers and half are unassigned. The unassigned ranges belong to the "fallback server," the server with the largest capacity.

We keep a list of random functions of the form $f(x) = ax + b \pmod{n'}$, and there are $L = \log_2 n' + f$ functions, where $f \geq 0$ is a number chosen to ensure even distribution. (Higher values of f constrain the distribution to be increasingly even, but increase the computational cost of the SIEVE function.) The algorithm in Figure 2 yields the mark for a connection:

Theorems proposed and proven in [3] include guarantees about the maximum amount of storage required for this hashing algorithm, the maximum unevenness of the distribution of requests across servers, and the efficiency of the algorithm. These assure that within a small number dependent on n and f as defined above, each server 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 server in the system will route connections to that server, thus avoiding as many timed-out connections as possible and keeping response times low.

Pang et al. [10] use their DNS measurements to evaluate and discuss the impact of federated deployment models on future systems, such as Distributed Hash Tables.

3.3 Marking implementation

A small userspace daemon runs on each server in the HashNAT system, checking its load every second. Load here is defined as number of TCP connections on a server which are in the TIME_WAIT state. The current capacity is calculated as the base capacity of the server minus a function of the load, because heavily loaded servers have less capacity to serve further requests. The daemon then sends the capacity of the server to every other server in the system, over the network. The daemon also receives all of this data from the other servers in the system. If any server does not receive a capacity report from another server within two seconds of the last one, it marks the silent server as having a capacity of 0.

A pair of pseudo-files, `/proc/net/dpr/read` and `/proc/net/dpr/write`, is created by a loadable kernel module `dpr.o` (or `dpr.ko` on kernel 2.6). The daemon writes the received load data into `/proc/net/dpr/write`. The kernel module receives the load data written to it via the pseudo-file, and updates the table of server capacities in kernel space. This table is used by 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 server has changed, then, within two seconds the server sends its changed capacity to itself and the other servers. Every server recalculates the number of ranges belonging to the server with changed capacity, and updates its list of ranges to suit the new number of ranges owned by that server, in a concurrency-safe way. As requests come in, the SIEVE hash function uses the list of ranges as it stands.

A script written in Perl sets up all of the loadable kernel modules and firewall rules necessary for the system to work. It then checks the server capacity table by reading the file `/proc/net/dpr/read` every two seconds. If the number of lines in the file (thus, servers in the system) increases, the script automatically adds rules to `netfilter`'s NAT rule table such that packets marked with the new table entry number will be routed to the new server, as described in section 3.1.

4. Testing

4.1. Method

Four Dell OptiPlex GX260s (the servers) were connected to a 100Mbps Ethernet network, and set up to run Linux on kernel version 2.6.8, with the additional modules described above. In addition to the normal complement of background processes, they also ran the `thttpd` http server and the userspace daemon described above. Each server had a file `/var/www/index.html` which contained 14139 bytes

of random data.

Six Dell OptiPlex GX260s (the testing clients) were connected to the same network, and set up to run a custom-written testing script which, upon command, would send HTTP 1.0 requests to a given server for a given time period, waiting small amounts of time between each request, and return the number of requests successfully completed. The testing script could also perform a dry run, i.e. going through the test, but without making the actual TCP connections and HTTP requests.

Another server outside the testing lab (the test runner) was scripted to send testing commands to the testing clients, and record the results, along with various indicators of server load sent from each server throughout the duration of the tests. For testing, the clients chose random servers from a list given by the test runner instead of looking up names using round-robin DNS. The distribution of the random choices was constrained to be optimal by telling the clients to choose servers randomly until an optimal distribution was attained. By an optimal distribution we mean a distribution with minimal variance, e.g., for six clients sending requests to two servers, three clients randomly choose one server and three choose the other. A non-optimal distribution would occur if five clients chose one server and one client chose the other. We will refer to this optimal distribution as the totally load balanced configuration.

Each individual test was 30 seconds long. For each run of tests, the test clients were instructed to fire requests for each of 13 different inter-request delay times, so as to present an increasing number of requests per second to the servers. In between each test in the run, the clients performed an identical dry run. There were eight test runs in all. First the six clients were all pointed at one server, with no distributed packet rewriting, then the six clients were split between two servers, then three, then four. This pattern was repeated with the servers connected using the HashNAT system.

4.2. Capacity functions

Section 3.3 leaves undefined the function which yields the capacity of a server. In the following results, the function in question was defined as $g(x) = Ce^{-kx}$ with $C=1048576$ and $k \approx 0.00069$ (such that $g(0)=1048576$ and $g(20000)=1$), with values less than 1 truncated to 1. x was defined to be the number of TCP connections on this server which are in the TIME_WAIT state (such connections have been closed in the last 120 seconds, subject to TCP/IP stack configuration). The reason for choosing our particular value of C is that it allows for a comfortable maximum of 4096 servers in one HashNAT system to be reached before overflows will occur when the capacities of all the servers are summed in the hashing code. As for k , the largest number of TCP connections in the TIME_WAIT state observed during testing was around 18,000. k was chosen commensurate to

that number, such that the value of $g(x)$ would not decrease too slowly or quickly as the servers are loaded down.

As has been noted, the capacity function is calculated not in the kernel, but in user space; this makes it quite easy to calculate the dynamic capacity of a server using other load metrics. This is left to future research.

4.3. Results

The throughput obtained using the HashNAT system vs. totally load balanced web servers is summarized in Figure 3, in which the “theoretical” data rate is obtained by multiplying the number of dry-run requests completed in each testing run by the number of megabits sent and received for each request (≈ 0.10871 Mbit) and dividing by the test duration (30 seconds), and the “real” data rate is the number of real requests completed in each testing run, multiplied and divided the same way. The four graphs in Figures 3 and 4 correspond to different numbers of servers (out of the total number) to which requests

were sent in different tests. A greater number of these means that the requests are more balanced between servers before the servers get them (which reduces the advantage of load balancing), and the aggregate bandwidth available for filling requests is greater (which raises throughput whether load balancing is present or absent). The totally load balanced web servers receive the optimal distribution described in section 4.1. The requests to the servers are balanced, they are processed locally and no routing is needed. Because of their minimal request processing, totally load balanced web servers form an upper bound on network throughput.

In Figure 3, upper-bound throughput (indicated by solid lines) with one server receiving requests results in the other three servers sitting idle (upper-left graph). With all four servers running the HashNAT system (dotted lines in the graph), but only one of the servers receiving requests, the server receiving requests has to either serve them or route them to other servers. If it were to serve all the requests, it could do no better than the upper-bound throughput. But in this test, the server receiving the requests routed all of them away from itself, because the fallback server in the calculation of the hashing function was another server. The servers in this test

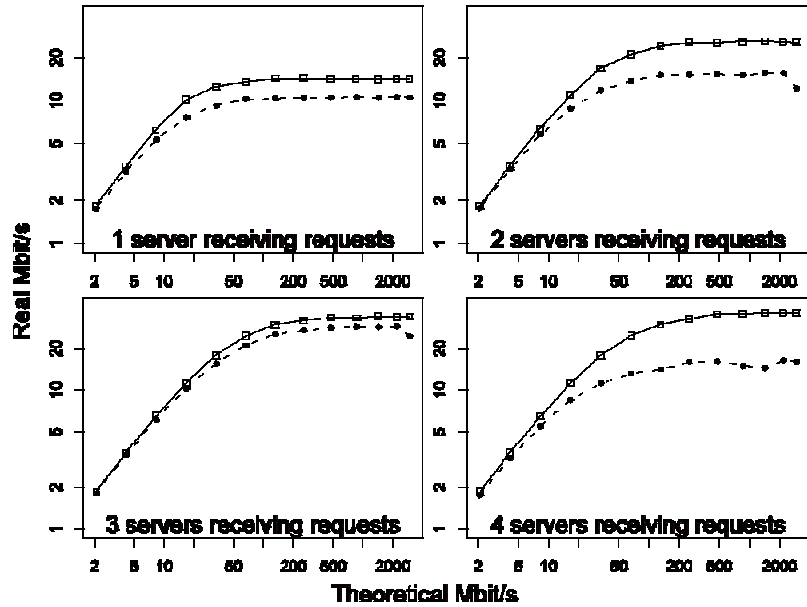


Figure 3: Comparison of throughput between distributed web servers using HashNAT (dotted lines) and the upper bound configuration (solid lines) with different numbers of servers receiving the total volume of requests.

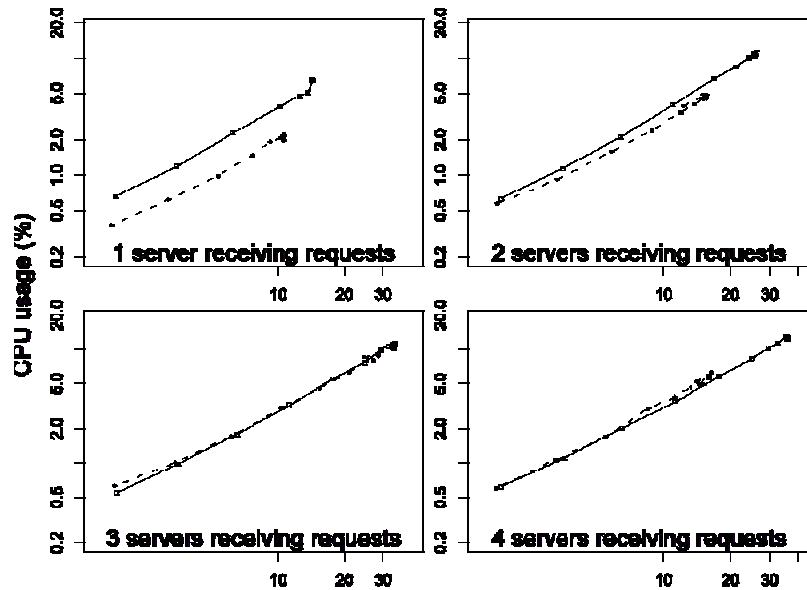


Figure 4: Comparison of CPU usage during tests between distributed web servers using HashNAT (dotted lines) and the totally load balanced configuration (solid lines) with different numbers of servers receiving the total volume of requests. The x-axis is real throughput in Mbit/s.

were on the same network with the clients, so in routing the data, the receiving server had to transmit and receive it twice over the same network. This accounts for the reduction in throughput. This is the degenerate case of any distributed

packet rewriting system: the requests are served on the other servers, and the one receiving requests is the router.

With two servers receiving requests, the upper-bound throughput predictably almost doubles (at the fastest theoretical data rates, the factor is around 1.8). Using the HashNAT system, the throughput goes up (around 1.4 times the throughput for one server receiving requests) but not as far up, again due to routing over the same network. Also, during the test, one of the servers receiving requests routed most of its requests to the other server receiving requests and not to one of the idle servers, so one server of the four remained idle.

With three and four servers receiving requests, the upper bound grows more slowly, because the switching fabric approaches saturation. With HashNAT enabled and three servers receiving requests, the system is at its fastest. This is because only 12,000 of the 40,000 requests made during the test were forwarded away from the servers which received them, so less data was sent over the same network twice.

Figure 4 shows the mean CPU usage of the four servers during the tests. The most dramatic difference between the totally load balanced configuration and the use of HashNAT is the case when one server receives all of the requests. While in this case it does not attain as high a data rate, it is clear that the mean CPU usage over all four servers is about 50% lower at the data rates it does attain. The advantage we see here is because the one host that received the requests spent more time in the kernel as opposed to any user process (since it forwarded all the connections made to it), and the kernel is smaller and more efficient than any application. When this very low CPU usage is combined with the CPU usage associated with serving some of the requests, which is observed on each of the servers to which the requests are routed, the resulting mean is lower. With two, three and four servers receiving requests, the requests, and thus the load associated with serving them, are spread more evenly among the servers, reducing the difference between the use of HashNAT and the totally load balanced configuration, as has been noted at the beginning of this section. But the CPU usage of the HashNAT system is never worse than the CPU usage of the totally load balanced configuration that corresponds to the upper-bound throughput configuration.

5. Conclusion

We have developed a new system called HashNAT, which can enable the use of cheap commodity hardware (or even embedded hardware) for large-capacity web serving. Servers can be added to the system at any point with no downtime at all. In case of a server outage, the other servers immediately avoid routing connections to the ailing server, minimizing request timeouts. The IP tunneling of DPR (and the cost it incurs of encapsulating and decapsulating IP packets) is replaced with faster and more common network address translation. Our preliminary tests are promising.

They show that the throughput of our distributed web system in the best case is very close to the upper-bound throughput, while the CPU usage on the servers is dramatically reduced.

References

- [1] E. Anderson, D. Patterson, and E. Brewer. "The Magicrouter, an application of fast packet interposing." <http://www.cs.berkeley.edu/~eanders/projects/magicrouter/>, 1996.
- [2] L. Aversa and A. Bestavros. "Load balancing a cluster of web servers using distributed packet rewriting." *IEEE International Performance, Computing and Communication Conference*, 2000.
- [3] A. Brinkmann, K. Salzwedel and C. Scheideler. "Compact, adaptive placement schemes for non-uniform requirements." *Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures*, 2002.
- [4] V. Cardellini and E. Casalicchio. "The state of the art in distributed web servers." *ACM Computing Surveys*, Vol. 34, No. 2, 2002, pp 263-311.
- [5] E. D. Katz, M. Butler and R. McGrath. "A scalable HTTP server: the NCSA prototype." *Computer Networks and ISDN Systems* 27, 2002, pp. 155-164.
- [6] A. Mourad and H. Liu. "Scalable web server architectures." *IEEE Symposium on Computers and Communications*, 1997, pp. 12-16.
- [7] P. Russell. "Netfilter architecture." <http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-howto-3.html>, 2002.
- [8] C. Wang. "A distributed Linux cluster server system with fault-tolerant ability." http://etdncku.lib.ncku.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0731102-115134, 2001.
- [9] R. Zhang, T. F. Abdelzaher and J. A. Stankovic. "Efficient TCP Connection Failover in Web Server Clusters." *23rd Conference of the IEEE Communications Society*, 2004.
- [10] J. Pang et al., "Availability, usage, and deployment characteristics of the domain name system", Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, Oct. 2004