On the Performance of TCP Splicing for URL-aware Redirection

Ariel Cohen Sampath Rangarajan Hamilton Slye
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
acohen@research.bell-labs.com

Abstract

This paper describes the design, implementation and performance of a layer-7 switch which supports URL-aware redirection of HTTP traffic. Currently, there are several vendors who are beginning to announce the availability of such switches in the market, but little or no implementation and performance information is available. We discuss design issues pertaining to such switches through a prototype implementation of a URL-aware switch in the Linux kernel, and analyze the performance of our implementation. We investigate the use of TCP splicing as a mechanism for improving the performance of the switch; we explore whether TCP splicing will benefit URL-aware redirection even though HTTP connections, on average, are short-lived and transfer small amounts of data. Results from our implementation show that TCP splicing does improve the performance of URL-aware switches that handle short-lived HTTP connections. Our results also re-affirm earlier findings that TCP splicing substantially improves the performance of any application-layer proxy when large amounts of data are transferred through the splice.

1 Introduction

URL-aware redirection (also known as "content-smart switching" [ArrowPoint]) refers to the capability of a switch located in front of clients or servers to redirect HTTP requests to servers based on the URL specified by the client in its GET request. When a user enters a URL into a browser, the browser constructs an HTTP GET request which contains the URL and other HTTP client header information. With URL-aware redirection, a switch located on the path between the client and the servers will intercept the request and use the information within that request to make a decision about the server to which the request should be directed. All this happens transparently to the client.

A number of products currently available perform basic layer-4 switching [Alteon, Foundry] which involves redirecting traffic based on transport-layer (TCP) information such as TCP ports as well as network-layer (IP) information such as IP addresses. Some uses of these products are: redirecting web traffic to caches (to facilitate transparent caching), server load-balancing, and fault-tolerance. URL-aware redirection at the switch further expands the scope of information utilized by the switch to layer-7 (application) information. The benefits derived from this enhancement include the ability to direct requests for particular kinds of content (such as images or video) to servers which are optimized for delivering that content, the ability to direct requests for dynamic content to live servers instead of caches, and the ability to reduce the need for replication in environments where content is replicated among servers for load-balancing and fault-tolerance—URL-aware redirection makes it possible to use partial replication instead of full replication. Looking beyond just the URL, it is also possible to parse the "cookie" information that is carried as part of the HTTP header and make redirection decisions based on this information.

Basic layer-4 switches that provide functionality such as load balancing and support for transparent caching perform TCP traffic redirection by redirecting the initial SYN packet from the client to the chosen destination and redirecting all subsequent packets on the connection to the same destination. To accomplish such redirection, the switch needs to "peek" into the IP and TCP headers to find connection information such as IP addresses and TCP port numbers. Based on this information, the switch uses mechanisms such as NAT (network address translation) and PAT (port address translation) to redirect the connection.

Redirecting TCP traffic based on application-layer

(layer-7) information is not as simple to accomplish. For any TCP transaction, application-level information is not available until the TCP connection establishment phase has been completed. This means that connections cannot be redirected at a switch by simply peering into a SYN packet as is possible with basic layer-4 switches. The TCP connection request from the client needs to be accepted at the switch and the connection established between the client and the switch before any application-level information can be received. Once the application-level information is received, this information is parsed to determine which back-end server or cache should receive the request, and the request is redirected.

One approach for redirection is the *TCP gateway* approach where another TCP connection is established between the switch and the back-end server, the client request is passed to the server through this connection, and the response is received at the switch from the server on this connection and transferred through the other connection to the client. Another approach would be to move the switch-side endpoint of the client-switch TCP connection to the server, thereby establishing a direct TCP connection between the client and the server. This approach is a proprietary solution used by Resonate [Resonate1] and is referred to as the *TCP Connection hop* approach [Resonate2].

As far as we are aware, only ArrowPoint [ArrowPoint] and Resonate [Resonate1] provide URL-aware redirection solutions. ArrowPoint provides hardware switches (CS-100, CS-800) which are capable of performing URL-aware redirection, whereas Resonate's product (Resonate Central Dispatch) is purely a software solution. Vendors such as Foundry Networks, Alteon WebSystems and Nortel Networks together with IPivot have recently announced their intentions to deliver switches with URL-aware redirection in the near future [TechWeb1, TechWeb2]. ArrowPoint uses the TCP gateway approach to support URL-aware redirection, and we believe that other vendors that have announced products in this space will follow suit. Also, the TCP gateway approach is more general than the TCP connection hop approach as the TCP/IP stack at the back-end servers needs to be extended in the latter approach. This is impractical, for example, when the switches are used on the client side to determine whether a client request should be redirected to a nearby cache or to a remote origin server based on the content that is requested. Hence, we focus on the TCP gateway approach in this paper. In the next section, we consider URL-aware redirection in more detail and discuss the motivation for our work.

2 URL-Aware Redirection

Let us now consider a specific example of the functioning of a URL-aware switch that uses the TCP gateway approach. In this example, an HTTP request from a client to an origin server is transparently "peeked" into by a URL-aware switch, which then either forwards the request to the origin server or to a nearby cache depending on the object that is being requested. For example, if the request is for a non-cacheable item, it will be forwarded to the origin server. Normally, when a client makes an HTTP request, a TCP connection is first established with the server. The client then sends information pertaining to the object it requires as part of a GET request[1]. The server parses this request and returns the requested object to the client. When a URL-aware switch is introduced to transparently intercept client requests, the switch intercepts the connection request packets from the client (based on the destination port being port 80), and sends them up to a local application-level proxy which understands HTTP. A TCP connection is established between the client and the proxy (which now masquerades as the origin server to which the client made the request). The client then sends the GET request to the proxy, which determines the destination (in this case, the origin server or a nearby cache) based on the requested object. The proxy establishes a TCP connection to the destination, forwards the GET request on this connection, receives the response and transfers it to the client.

The TCP gateway approach exacts an overhead due to the use of two TCP connections. Data that passes from the server to the client needs to go all the way up the protocol stack to the application layer at the switch and then again down the protocol stack when it is put back on the connection to the client. To improve the performance of application-layer proxies which function as TCP gateways, TCP splicing has been proposed as a solution [Maltz1]. In this approach, once the two TCP connections are established, they are "spliced" together so that IP packets are forwarded from one endpoint to the other at the network layer without having to traverse the TCP layer to the application level on the switch. This requires that appropriate address translations and sequence number modifications be performed on

[1] There are other request tokens such as POST and HEAD, but without loss of generality we will only refer to GETs in the rest of the paper.

the packets. For example, packets arriving on the connection from the server to the switch that are to be forwarded to the client, should be translated so that the addresses and sequence numbers on these packets match the ones that would be found on the corresponding packets if the application-layer proxy received this data and then put it back on the TCP connection from the switch to the client.

In [Maltz1, Maltz3], the use of TCP splicing has been studied to improve SOCKS proxy performance. Performance numbers provided in [Maltz1] show TCP splicing to be beneficial when large amounts of data are transferred from the servers to the clients. Further, results in [Maltz2] show that HTTP caching proxy throughput can be improved with TCP splicing, again with big chunks of data transfer. HTTP caching requires data to go up the application layer to be cached anyway and may not see the full benefits of TCP splicing. Also, for a URL-aware switch that handles short-lived HTTP connections, where the number of control packets in establishing the two TCP connections may be as large as the number of data packets, it is not apparent that the benefits gained by splicing the data packets at the network layer are not offset by the extra penalty paid in snooping on the control packets to register connection state information. Hence, useful conclusions as to how a URL-aware switch will benefit from TCP splicing cannot be reached from these results. It appears that the URL-aware redirection solution from ArrowPoint makes use of TCP splicing but there is no documented data from ArrowPoint or other potential vendors whether TCP splicing benefits URL-aware redirection.

The design, implementation and performance of a hardware-based URL-aware switch called L5 are described in [Apostolopoulos]. L5 consists of programmable port controllers connected to a switch core and a general-purpose CPU. TCP splicing is used in this switch to take the CPU out of the data path once the URL is obtained and a connection to the server is established. After splicing, all packet processing is handled by the port controllers. In our software-based switch, all processing is done by the CPU since port controllers are not available. The goal of splicing in this case is to take the user-level proxy and TCP out of the data path and perform all packet processing after splicing within the kernel at the IP level.

Our contributions in this paper are as follows. First, we discuss in detail the design and implementation of a URL-aware switch in the Linux kernel that uses TCP splicing. Then, using performance measurements from our implementation, we show that TCP splicing does benefit URL-aware redirection even for small TCP sessions that transfer as few as 1 KB of data through the splice. Further, using workloads representing different object sizes and connection durations, we re-affirm the conclusions reached in [Maltz1] that when large amounts of data are transferred, TCP splicing provides substantial performance benefits. In the next section, we present implementation details of the URL-aware switch. Section 4 presents performance results from our implementation. Conclusions are presented in Section 5.

3 Implementation Details

There are four components to our switch implementation which is shown in Figure 1. The first component is an application-level proxy (*proxy-s*) which accepts TCP connections from the clients, parses the GET requests, determines the destination server and establishes a connection to that server thereby enabling URL-aware redirection. The second component is a loadable kernel module implemented inside the Linux kernel which we will refer to as the splice module (*sp-mod*). *sp-mod* monitors packets exchanged on the two connections and maintains the state machines that represent the two connections before the connections are spliced. It also maintains the state machine for the spliced connection. When appropriate, *proxy-s* communicates with *sp-mod* and indicates the pair of connections that need to be spliced. The third component is another loadable kernel module which we refer to as the NEPPI (Network Element for Programmable Packet Injection) module. This module performs low-level header manipulation operations such as network address translations and sequence number translations on the packets. When *sp-mod* receives splicing requests from *proxy-s*, it interprets those requests into low-level header manipulation instructions which it then sends to the NEPPI module. The fourth component is the Linux *ipchains* firewall, which we use to filter packets. The *sp-mod* module instructs NEPPI to filter appropriate packets to be processed. These instructions are translated by NEPPI into appropriate system calls to the *ipchains* firewall. Each of these components is described in more detail below, starting with the NEPPI and *sp-mod* modules.

3.1 The NEPPI Module

In our earlier work [Cohen], we developed a software substrate implemented as a Linux kernel mod-

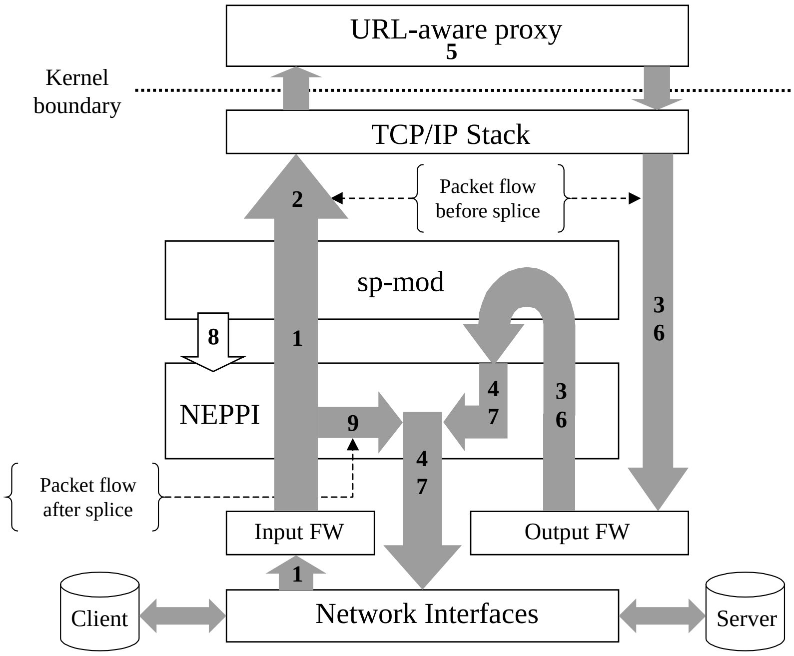

Figure 1: URL-aware switch architecture

ule (NEPPI) that provides a set of APIs which can be used to obtain, manipulate, and inject IP packets. These APIs are typically used by *gateway programs* to provide higher level services. Gateway programs can either run as user-level processes or as kernel modules themselves. Gateway programs send NEPPI rules that specify the properties of packets on which they wish to operate. Such rules may include IP address ranges for the source and destination, TCP/UDP port number ranges, etc. Based on the rules obtained from the gateway programs, NEPPI generates packet filter rules which are sent to the Linux packet filter. An arriving packet which triggers a rule is sent from the packet filter to NEPPI which either sends it to the requesting gateway program, or manipulates it in accordance with a manipulation rule specified by the requesting gateway program. Such manipulation rules include address translations, TCP sequence number changes, and TCP window size changes. Packets generated or modified by the gateway programs are returned to NEPPI, which in turn modifies them further (if they match a manipulation rule) and injects them into the network. In the case where the destination address of a packet is the switch itself, NEPPI will hand the packet to the protocol stack instead of to the network interfaces. One of our goals in designing NEPPI was to provide an architecture where packet header manipulations which are generic and which occur on a large number of packets are performed at NEPPI, whereas more specialized packet manipulations (including payload modifications) which occur only on isolated packets are performed at the gateway programs. This way, the architecture will provide a "fast path" for most packets (the ones processed by NEPPI), and a "slow path" for only a few packets (the ones processed by the gateway programs).

In the implementation of the URL-aware switch, the NEPPI module serves as the low-level packet processing substrate which is used by the *sp-mod* gateway program to perform low-level packet manipulation functions. A useful feature provided by NEPPI is support for redirecting packets to a local TCP port. Using this feature, our URL-aware switch can operate in transparent mode. Client requests destined towards origin servers are transparently intercepted and redirected to the URL-aware proxy, *proxy-s*. In this case, the *sp-mod* module requests NEPPI to redirect all incoming packets from clients to a local TCP port on the switch in which *proxy-s* is listening. NEPPI sends these packets to the local protocol stack even though the destination IP address on these packets is not that of the switch. *proxy-s*, which listens on the local TCP port, will receive this data and will even be able to determine the original destination for the data. Details on how NEPPI is used by the *sp-mod* module are given in the next section.

3.2 *sp-mod*

The *sp-mod* module monitors all packets on the client-proxy connection as well as the proxy-server connection, and maintains the TCP state machine for these connections. It also monitors all packets after the splicing is done, and maintains the state machine for the spliced connection. It accomplishes this using NEPPI as follows. The Linux *ipchains* packet filter provides INPUT and OUTPUT firewalls. The INPUT firewall filters packets from outside coming in, while the OUTPUT firewall filters packets being sent out. *sp-mod* registers with NEPPI and instructs NEPPI to forward all packets that match the following rules: **a)** all packets from clients destined to port 80 (HTTP port). **b)** all packets from servers that originate at port 80 and are destined to the URL-aware proxy. Rules **a** and **b** are translated by NEPPI into Linux system calls to the INPUT firewall. **c)** all packets generated locally (by the URL-aware proxy masquerading as the origin server) destined to the client; these packets will originate at port 80. Finally, **d)** all packets generated locally by the URL-aware proxy destined to the servers; these packets will be destined to port 80. Rules **c** and **d** will be translated by NEPPI into Linux system calls to the OUTPUT firewall.

Let us now discuss the steps taken at the switch when a client request arrives. See Figure 1 in conjunction with the following description.

When a client makes an HTTP connection request to an origin server, the SYN packet is transparently intercepted by the INPUT firewall and sent to NEPPI, which in turn forwards it to *sp-mod* **(1)**. *sp-mod* records this new connection request information and the state of this connection. It then instructs NEPPI to forward this packet to the local URL-aware proxy port **(2)**. As mentioned earlier, NEPPI is capable of forwarding packets to a local port even though the packets do not carry the switch address as the destination address. The outgoing SYN-ACK packet is intercepted at the OUTPUT firewall and forwarded to *sp-mod* **(3)**. The connection state is updated at *sp-mod* and the packet is sent through NEPPI to the client **(4)**. In order for these packets not to be intercepted again at the OUTPUT firewall, NEPPI bypasses the OUTPUT firewall when it sends packets out. Note that these packets will carry the origin server's IP address as the source address since the URL-aware proxy masquerades as the origin server. When the ACK to the SYN-ACK is received, it is again intercepted at the INPUT firewall and forwarded to *sp-mod* **(1)**, which updates connection state and forwards it to the local port **(2)**. This completes the initial handshake.

After the initial handshake, the packet(s) containing the GET request are sent by the client to the server. Again, these packet(s) are intercepted at the INPUT firewall and forwarded to *sp-mod* **(1)**. At this time, *sp-mod* records the *seq* (**seq1**) and the *ack-seq* (**ack_seq1**) on the GET packet and updates the connection state before forwarding the packets to the local port **(2)**. The URL-aware proxy processes the GET request and selects a back-end server to redirect this request based on the contents of the URL **(5)**. For example, if the switch is placed in front of clients to transparently redirect requests to caching servers, an appropriate caching server will be chosen. Once the server is chosen, it initiates a TCP connection to that server. The SYN packet from the proxy to the server is intercepted at the OUTPUT firewall and forwarded to *sp-mod* **(6)**. Again, new connection information is recorded and the packet is forwarded to the server through NEPPI **(7)**. The SYN-ACK and ACK packets similarly pass through *sp-mod* and state information is updated. Note that *sp-mod* maintains state information for all connections but cannot identify which pair of connections should be spliced. Once the initial handshake between *proxy-s* and the server is completed, the proxy sends the GET request to the server. But before that, it sends a *splice* command to *sp-mod* with the appropriate endpoint information, instructing *sp-mod* to splice the connections together. The endpoint information includes two four-tuples, one each for each connection: the client IP address and TCP port number, the address and port number of the proxy endpoint on the client side, the address and port number of the proxy endpoint on the server side, and the server address and port number. When the GET packet(s) from the proxy to the server pass through *sp-mod*, it records the *seq* (**seq2**) and the *ack-seq* (**ack_seq2**) values on the GET packet.

sp-mod uses this pair of four-tuples it received from *proxy-s*, as well as the sequence number and ack-sequence number it recorded on the two connections, for the purpose of splicing connections together. When the first ack or data packet is received from the server in response to the GET request, *sp-mod* sends instructions to NEPPI to splice the connections together **(8)**. The instructions are in the form of address translation and sequence number translation instructions. The destination address and port number on the packets from the server to the proxy are changed to those of the client, and the *seq* and *ack-seq* numbers are re-mapped to those that would be found in corresponding packets if these packets were received by *proxy-s* and put on the TCP connection to the client. Similarly, address and sequence number translations are performed on the acknowledgment and other packets from the client before they are sent to the server. The source address and port number are changed so that the packet appears to the server as if it was sent by *proxy-s*. The *seq* and *ack-seq* numbers are re-mapped to the appropriate numbers to appear as if they were sent by *proxy-s* on its established connection to the server. The sequence number re-mapping is easily accomplished using the offsets $\delta_1 = \mathbf{seq1} - \mathbf{seq2}$ and $\delta_2 = \mathbf{ack\_seq1} - \mathbf{ack\_seq2}$, which are either added or subtracted to the sequence and ack-sequence numbers depending on the direction of packet flow.

Once the splice is done, all data packets from the server to *proxy-s*, and all acknowledgment packets from the client, are redirected to the client and server, respectively, at the network level **(9)**. As soon as the splice is performed, the two TCP endpoints on the switch bound to *proxy-s* (one on the connection to the client and one on the connection to the server) no longer receive packets and can be closed. To accomplish this, RST packets are generated by *sp-mod* masquerading as the client or the server (depending on which endpoint needs to be closed) and sent to *proxy-s*. When the data transfer has been completed, either the server or the client can initiate an active close. In either case, the FIN packets and the corresponding ACKs are still spliced so that the endpoints at the client and the server, each representing one endpoint of each connection, are also closed cleanly.

3.3 The URL-aware Proxy (*proxy-s*)

The URL-aware proxy *proxy-s* is a multi-threaded application-level program. A listener thread listens for incoming client connection requests and distributes these requests to a pool of worker threads. Once a client connection has been accepted and the GET request has been received, a connection is established with a chosen server. Before the GET request is sent to the server, *proxy-s* sends a *splice* command to *sp-mod* as described earlier. It uses the *getsockname* function call on the two socket endpoints to get the four-tuple information to be conveyed to *sp-mod*. As the client requests are transparently redirected to *proxy-s*, the *getsockname* function call on the client side socket endpoint will return the server's IP address and port number as the local endpoint. Library functions are provided which enable *proxy-s* to send the *splice* command to *sp-mod*.

The current implementation of *proxy-s* processes only the URL to make redirection decisions. The proxy can be configured to redirect requests based on extensions. For example, URLs with the GIF, JPG and HTML extensions can be redirected to different servers as shown in Figure 2. Also, CGI requests can be redirected to specified servers. A primitive form of cookie processing is available, where requests can be redirected to a server based on the presence or absence of a cookie. This will be useful, for example, when the switch is placed as a front-end to a set of caching servers. In this case, the proxy can be instructed to forward all requests that contain a cookie to the origin server instead of redirecting them to caching servers. As described in [Apostolopoulos], other HTTP request headers can be processed to provide various functionalities. As our current focus in on TCP splicing, we have not concentrated on this aspect.

3.4 *ipchains* firewall

As mentioned earlier, we use the existing Linux firewall functionality to filter packets on the *INPUT* and *OUTPUT* firewall chains. *ipchains* provides the capability to specify packet filtering rules based on source IP and destination IP address ranges, source TCP and destination TCP port number ranges, and protocol numbers. These rules are specified using system calls to the firewall. NEPPI sends commands to the firewall to filter packets requested by *sp-mod*. For each rule, a *mark* can be specified which is used by *ipchains* to tag the packets when they are filtered and forwarded. *sp-mod* uses these marks to distinguish the direction of flow for each packet so that appropriate processing can be performed. This removes the extra overhead which would be required if *sp-mod* had to look at the packet headers to determine the direction of flow.

During the connection establishment phase from the client to *proxy-s*, the packets from *proxy-s* to the client are intercepted at the OUTPUT firewall and redirected to *sp-mod*. Packets from *proxy-s* to the client are only monitored by *sp-mod* to maintain the state of the connection. It does not instruct NEPPI to perform any header modifications on these packets. This means that after *sp-mod* looked at these packets, and NEPPI received them and sent them on the network, these packets would normally pass again through the *OUTPUT* firewall, filter rules would again get fired and the packets would be forwarded back to *sp-mod*. This would create an infinite loop. A similar problem exists on the connection from *proxy-s* to the server. To get around this problem, we had to change the raw socket imple-

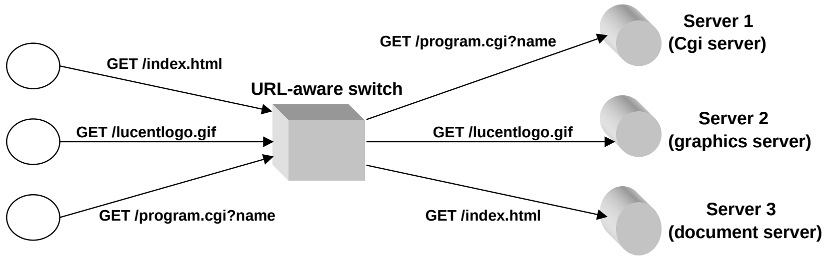

Figure 2: URL-aware redirection

mentation that NEPPI uses to send packets on the network so that the OUTPUT firewall is bypassed.

Finally, as discussed in [Maltz1], when two TCP connections are spliced, it is important that the TCP options on the two connections be adjusted and made compatible. In our current implementation, we address this problem by simply removing all options except the Maximum Segment Size option on the SYN packets on both connections.

4 Performance

We conducted three experiments to study the performance impact of TCP splicing in our Linux-based switch for URL-aware redirection. All experiments consisted of running an HTTP workload generator on five clients. The first experiment involved accessing Web sites within the Lucent Technologies corporate network; the second experiment consisted of accessing external sites, and the third experiment involved sites on the local network. For the purpose of the first two experiments, all HTTP GET requests were directed by the switch to the server specified by the client (i.e., no URL-aware redirection was performed.) The third experiment made use of the URL-aware redirection capability of the switch. We ran each workload with two versions of the proxy software running on the switch: one which performed TCP splicing (*proxy-s*), and one which did not utilize TCP splicing (we refer to this version as *proxy-ns.*) In both versions, the proxy obtains a GET request from a client, opens a TCP connection to the appropriate server, and sends the GET request to the server. At that point, *proxy-s* requests *sp-mod* (which resides in the Linux kernel) to splice the TCP connection between the proxy and the client with the TCP connection between the proxy and the server. From that point onward, *proxy-s* does not have to handle any traffic on these connections. *proxy-ns*, on the other hand, does not splice the connections and hence has to forward data between its client-side socket and its server-side socket at the application level.

We used a locally-developed HTTP workload generator called WebWatch in our experiments. This program generates multiple concurrent HTTP requests and measures the obtained performance. WebWatch reads a file containing a list of URLs and a file containing various parameters such as the desired number of concurrent requests and the number of passes through the URL file. Once WebWatch starts executing, it starts sending HTTP GET requests based on the URLs in its input file. The specified URLs are obtained along with all their embedded documents. WebWatch issues a certain number of concurrent requests (as specified in its parameters file), and then waits for the arrival of the responses or the expiration of a timer. Once all the data is received (or the timer expires), a new batch of requests is issued.

All the experiments presented here were run for a period of three minutes with a concurrency setting of 75 (i.e., 75 HTTP requests were issued at a time.) *proxy-s* and *proxy-ns* were run with a thread pool of size 30. All clients were PCs based on Pentium II 400MHz CPUs. The switch was a PC based on a Pentium III 550MHz CPU. The experiments were run multiple times to determine whether there was a significant variation among the results of different runs. The results of all runs were very similar to the results of the runs reported here.

The workload for the first experiment consisted of Web sites within the Lucent Technologies corporate network. The clients and the switch were on a Fast Ethernet network which is connected to multiple regular Ethernet networks through a router. Table 1 shows the results of this experiment. The "Conn" column shows the number of HTTP GET requests from the workload which WebWatch was able to issue within the period of the experiment. Note that the number of TCP connections for which the switch serves as an endpoint is actually twice the number which appears in the "Conn" column

Program	Conn	Util
proxy-s	13,194	1.52%
proxy-ns	7,917	5.13%

Table 1: Results for the internal workload

Program	Conn	Util
proxy-s	8,250	1.18%
proxy-ns	5,594	2.94%

Table 2: Results for the external workload

since each GET request from a client results in establishing one TCP connection between the client and the switch and another TCP connection between the switch and the server. The "Util" column shows the average CPU utilization at the switch for the period of the experiment. We see that using *proxy-s* resulted in a 67% higher number of connections than *proxy-ns,* while incurring a significantly lower CPU utilization. With *proxy-ns*, connections take longer to complete, so WebWatch was able to complete a smaller number of connections with *proxy-ns* than with *proxy-s* in the given amount of time. Recall that for each client, WebWatch was configured to submit a batch of 75 requests, and then wait for them to complete before sending the next batch. Since there is much unused CPU time, we could potentially increase the number of clients used with *proxy-ns* until the number of connections achieved matches the number obtained by *proxy-s* with just five clients. This would, however, result in an even larger difference in CPU utilization between *proxy-s* and *proxy-ns*.

The workload for the second experiment consisted of external Web sites (various news sites). Table 2 shows the results of this experiment. Again, we see that using *proxy-s* resulted in a significantly higher number of connections (47% higher) and lower CPU utilization.

To observe the impact of using TCP splicing on different file sizes, we ran a third set of experiments. The workload for this set of experiments consisted of retrieving files of a uniform size from three HTTP servers on the same Fast Ethernet network as the clients and the switch. Files had three possible extensions; the URL-aware switching capability of the switch was configured to direct each possible extension to a different server. We ran seven experiments for seven different file sizes. Table 3 shows the results of this set of experiments. File sizes are in bytes. Similar to the experiments described previously, the workload was run for a period of three minutes. The "Conn/s" column shows the average number of connections obtained per second. The table shows the significant increase in the improvement obtained by *proxy-s* when compared to *proxy-ns* as the file size increases.

proxy-s shows the biggest performance gain for large transfers. This is a result of the following: for small transfers, the overhead associated with setting up the connection and issuing the GET request is high compared to the overhead associated with transferring the data itself. This overhead is similar for *proxy-s* and *proxy-ns*. When performing large transfers, however, *proxy-ns* is strongly affected by the need to move the data between the sockets at the user-level proxy. *proxy-s*, on the other hand, does not need to handle the data at the user-level proxy at all since the two TCP connections are spliced together within the kernel.

It is important to note another advantage which *proxy-s* has when compared to *proxy-ns*. After the splicing of the two TCP connections (client to proxy and proxy to server) is accomplished, the connections are reset, so they cease to exist. When no splicing is used (i.e., in the case of *proxy-ns*), the connections exist for the duration of the data transfer and for an additional period beyond the time when the connections are closed (the 2MSL timeout required by TCP). This results in a much larger number of TCP connections with state in the protocol stack when running *proxy-ns*.

As seen in Table 3, the CPU in the switch is fully loaded when running *proxy-ns*. We were not able to fully load the CPU when running *proxy-s*. For files of size 10,000 bytes and larger, the 100 Mbit/sec network was fully utilized. For smaller files, neither the network bandwidth nor the CPU utilization was a bottleneck. Hence, one might expect to obtain larger numbers of connections for those files than the numbers we obtained. We are not certain about the reasons for not obtaining larger numbers. Possibly, a limitation of the servers was reached, or perhaps some limitation of the Linux TCP/IP stack was reached. We plan to investigate this issue further in the future.

Empirical data indicates that the average web object size is around 10 KB [Lee]. For objects of this size, Table 3 shows that TCP splicing results in a 53% increase in the number of connections along with a 38% decrease in CPU utilization.

File size	Splice *(proxy-s)*			No splice *(proxy-ns)*		
	Conn	Util	Conn/s	Conn	Util	Conn/s
1,000	233,035	61.32%	1,295	183,307	95.49%	1,018
2,000	227,107	64.85%	1,262	172,984	93.61%	961
5,000	214,782	71.20%	1,193	153,202	97.99%	851
10,000	177,216	61.84%	985	116,071	99.82%	645
20,000	89,477	21.24%	497	82,866	99.56%	460
30,000	62,907	13.42%	349	60,939	96.80%	339
300,000	6,843	1.60%	38	5,455	99.45%	30

Table 3: Results for a uniform workload of different file sizes

5 Conclusions

The implementation of a URL-aware redirection switch was presented. The switching functionality is provided by loadable kernel modules for the Linux OS along with a user-level proxy. The user-level proxy serves the purpose of accepting connections from clients, determining the requested URLs, making decisions about the appropriate servers, and submitting the requests to the servers. Upon submission of a request to a server, the proxy removes itself from the data path by requesting the kernel to splice at the IP level the connection between the client and the proxy with the connection between the proxy and the server [Maltz1, Maltz2]. The performance impact of TCP splicing was studied by comparing the performance of the switch with and without splicing on a variety of workloads. TCP splicing resulted in a significant performance improvement on all workloads. As expected, the effects of TCP splicing are particularly striking for large requests, but our results show significant performance gains even for workloads consisting of small requests of one kilobyte.

Acknowledgments: We would like to thank Reinhard Klemm for providing us with his HTTP workload generator WebWatch.

References

[Alteon] ACEdirector, *http://www.alteon.com/products.*

[Apostolopoulos] Apostolopoulos, G., V. Peris, P. Pradhan, and D. Saha. *L5: A Self Learning Layer-5 Switch.* IBM Research Report 21461, 1999.

[ArrowPoint] "The content smart internet", *http://www.arrowpoint.com/solutions/whitepapers/CSI.asp.*

[Cohen] Cohen, A., and S. Rangarajan. A Programming Interface for Supporting IP Traffic Processing. In: *Proceedings of the 1st International Working Conference on Active Networks (Lecture Notes in Computer Science, Vol. 1653, Springer),* pp. 132–143, 1999.

[Foundry] "ServerIron server load balancing switch", *http://www.foundrynet.com/serverironsepc.html.*

[Lee] Lee, R., and G. Tomlinson. Workload Requirements for a Very High-Capacity Proxy Cache Design. In: *Proc. of the 4th International Web Caching Workshop (NLANR/CAIDA),* 1999.

[Maltz1] Maltz, D.A., and P. Bhagwat. *TCP Splicing for Application Layer Proxy Performance.* IBM Research Report RC 21139, 1998.

[Maltz2] D.A. Maltz and P. Bhagwat. *Improving HTTP Caching Proxy Performance with TCP Tap.* IBM Research Report RC 21147, 1998.

[Maltz3] D.A. Maltz and P. Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. In: *Proc. of IEEE INFOCOM '98,* pp. 1037–1045, 1998.

[Resonate1] "Resonate Products—Central Dispatch", *http://www.resonate.com.*

[Resonate2] "Resonate Central Dispatch TCP Connection Hop", *http://www.resonate.com/products/tech_overviews.html.*

[TechWeb1] "Switch Vendors Detail Enhancements", *http://www.techweb.com/wire/story/TWB19990316S0006,* 1999.

[TechWeb2] "Portals Provide Layer 4 Acid Test", *http://www.techweb.com/wire/story/TWB19990315S0009,* 1999.