Fully Pre-Splicing TCP for Web Switches

Yeim-Kuan Chang, Wen-Hsin Cheng and Chung-Ping Young
Department of Computer Science and Information Engineering
National Cheng Kung University, Taiwan R.O.C.
{ykchang,cpyoung}@mail.ncku.edu.tw

***Abstract*—Fully pre-splicing (FPS) is an extension of TCP splicing which is content-blind that prevents the switches from using application layer information for forwarding decisions. FPS extends TCP splicing to support content-aware load balancing algorithms and pre-splices the client's connections to web servers. In addition, FPS extracts the application information in kernel-space for eliminating the cost of moving data twice through user/kernel protection boundaries and the latency of scheduling the processes. To achieve our design, we extended the TCPSP project of LVS using Linux Netfilter which defines specific hooks to provide a verdict for the packet. On the performance results, FPS improves the TCPSP throughput dramatically.**

Keywords: Fully pre-splicing, FPS, TCP splicing, TCP handoff, layer-7 web switch, content switch, pre-fork connection, distributed web server.

I. INTRODUCTION AND RELATED WORKS

A web site usually consists of a firewall, Layer2/3 switches, load-balancing devices, various kinds of cache and web servers. In addition to these devices and servers, applications such as Common Gateway Interface (CGI) programs, Personal Home Page (PHP), Active Server Page (ASP), and various Java technologies are used to generate dynamic web pages. A typical web site has become very complicated to manage because of the increasing performance requirements, a rich set of applications generating and storing dynamic web pages, and rapidly changing Internet technologies. One way to improve performance of a web site is to control how web requests and responses go in and out of the web site. The web switches can provide the highest level of flow control over the web traffic. The web switches look up the information in HTTP headers as well as TCP and IP headers to decide how the requests get served from the web site.

Traditionally it is the responsibility of proxy servers or web servers to redirect the web requests using URL, HTTP headers, or users' registration information. However, controlling how web requests and responses go in and out of the web site becomes the bottleneck because of the fast growth of Web services. Designing the web switch to offload the classification tasks based on user level information provides many benefits.

There are two characteristics for the layer-7 web switches. The first concerns the implementation layer: kernel-based or application-based. The second considers the flow of the packet traffic between the client and the web cluster. The main difference lies in the backward flow, because all inbound packets must pass through the web switch. In two-way architectures, the outbound packets pertaining to a response pass again through the web switch, the major technique called TCP splicing [1]. The TCP splicing takes care of all data forwarding operations directly in the kernel, leaving the set-up, tear-down and logging tasks specific to each type of proxy in the user level application that are easy to modify or amend as needed. It improves the current state of the art in three ways [3]: Performance: A proxy or firewall using TCP splicing acts like a layer 3.5 router; it does not incur either transport or application layer protocol processing overhead for each packet it processes. The reduced complexity and code path length dramatically improves throughput. Less book keeping: Proxies using TCP splicing need to maintain less TCP state information for each of the connections that pass through them, and the proxy does not have to buffer any packets. Better end-to-end semantics: TCP splicing enables two ends of the connection to communicate as peers, allowing control information to flow end-to-end. Aside from other advantages, this provides the connection with true TCP reliability semantics and correct urgent data handling.

The intuition behind TCP splicing is that we can change the headers of incoming packets as they are received and immediately forward them, rather than passing packets up through the protocol stack to user space, only to have them passed back down again. The effect is to have the proxy relay packets as if it were a layer 3.5 router. Authentication, logging, and other tasks are done by the proxy in user space as normal, but the data copying part of the proxy – where the performance is normally lost – is replaced by single ioctl() call to set up the splicing. After the splicing is initiated, the user level proxy can go on to other tasks.

The problem of TCP splicing, is its content-blindness such that only the first packet of the flow is used for load balance. The major purpose of our proposed system is to make TCP splicing to balance each packet in the flow. The other purpose is to reduce the latency of accessing web pages as much as possible. Our proposed system succeeds in eliminating the server side three-way handshake latencies by using fully pre-splicing methods.

The rest of the paper is organized as follows. Section 2 presents our proposed fully pre-splicing architecture and the prototype implementation of FPS on Linux kernel version 2.4.18. In Section 3, we describe the performance evaluation and compare the proposed FPS with TCPSP (TCP splicing) project [2] of the LVS (linux virtual server) [3]. In Section 4, we conclude our proposed system FPS.

II. THE PROPOSED FULLY PRE-SPLICING (FPS)

In this section, we propose a fully pre-splicing (FPS) technique by extending the TCP splicing technique and make the web switches to be content-aware. Content-aware switches can always use the application layer information contained in HTTP requests from clients to make forwarding decisions. After receiving the responses generated from web servers, the web switches reply them to clients by packet forwarding. FPS completes the dispatching work in kernel-space instead of user-space for reducing the unnecessary data copy latency between kernel and user space. FPS has been implemented as an extension of the TCPSP project of the LVS project. Therefore, FPS was developed as loadable kernel module of Netfilter [6] in RedHat 8.0 with Linux kernel 2.4.18.

A. *Architecture*

Layer-7 web switch establishes a TCP connection with the client and another TCP connection with a selected server. The switch forwards TCP data between these two TCP connections. This task requires TCP endpoint processing and application layer data forwarding. Web switching is much more complicated than that of IP routers or Layer 4 switches, which forward packets based on packet header information. Current application layer proxies suffer major performance penalties as they spend most of their time moving data back and forth between the two connections: context switching and crossing protection boundaries for each chunk of data they handle. TCP splicing provides kernel support for data relaying operations which can sustain a higher data throughput of normal proxies. Cohen [3] showed that TCP splicing using URL information results in a 67% higher number of connections than normal proxies.

TCP splicing that provides kernel support for data relaying operations which can improve a data transfer throughput of normal proxies. In HTTP/1.1, because TCP splicing makes load balancing only for the first request, following requests are distributed with no load balancing. Therefore, the load in each cluster server is unequal. Yang and Luo [4] proposed a Scalable Server Architecture approach; we named it SSA in this paper. SSA is similar to TCP splicing, but supporting sophisticated load balancing policies. In addition, SSA has also implemented with a pre-fork connection approach that web switch has pre-forked connections to web servers before the client connects to the switch. The pre-fork connection method reduces the latency of connecting the selected web server significantly.

The dispatching module also handles the persistent connections suggested by HTTP 1.1. The persistent connection uses one TCP connection to carry multiple HTTP requests, thereby reducing server load and client perceived latency. For HTTP 1.0 requests, the distributor can reuse the pre-forked connections to carry these requests, which will avoid extra TCP three-way handshakes and multiple slow-starts. Otherwise, the distributor splits multiple HTTP 1.1 requests within a persistent connection into single requests. If these requests belong to the same session, the dispatcher routes them to the server assigned to the same session. Else, it schedules the individual HTTP 1.1 requests to different servers based on content of theses requests.

But we argue that there are too many swapping times between the pre-fork connections and the mapping table. The client automatically sends subsequent requests to get all embedded objects for merging into a complete page. In SSA's approach which is based on Round Robin algorithm. The request "GET /a.jpg HTTP/1.1" may be dispatched to the different Server. In this case, the number of swapping between mapping table and available connection list is more once. In current www environment, the number of embedded objects of a page is most likely

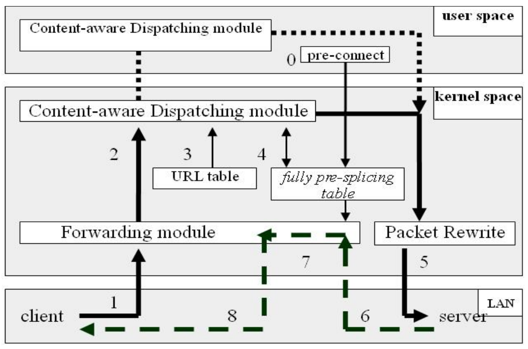


Figure 1. The packet flow of FPS

more than four. Therefore, the performance of SSA will become worse.

In our proposed FPS, pre-splices all the connections between the web switch and web servers, which resembles the SSA. The client establishes a TCP connection to web switch that splices the client's connection with one connection of each web server. The difference between FPS and SSA is the number of produced connection entries in the mapping table. Because of fully pre-spliced connection with each web servers, dispatching requests from the same client to the selected web servers will cost no extra swapping delay. The detailed steps of FPS are shown in Figure 1.

0) Web switch pre-forks several TCP connections to each backend server. These pre-forked connections are kept in a long time and put into the available connection list.
1) After the TCP three-way handshake between the client and the web switch is completed, the client requests the required page, the same as TCP splicing and SSA.
2) The packet will never go to the upper application layer. The packet is sent to the dispatcher module for making server decisions.
3) In this state, the dispatching module parses the HTTP request in the kernel space and makes a decision to send the request to the selected server according to the URL table.
4) Once the dispatcher selects a target server, it also chooses a pre-forked connection from the available connection list connecting to each server and splices them. Then, the dispatcher stores related information about acknowledge number, sequence number, timestamp, and TCP options in the fully pre-splicing table. The connection table adds new entries with the number of cluster servers while one client's connection comes.
5) After the protocol header modifications of the packet are finished, the packet is sent to the real server for serving this HTTP request.
6) The server receives forwarded request from switch, acknowledges receipt of packet and begins to process it. The server replies the response of the request to the switch.
7) Because the server IP and server Port has been hashed in the fully pre-splicing table, in Step 4, the response from the server will redirect to the client

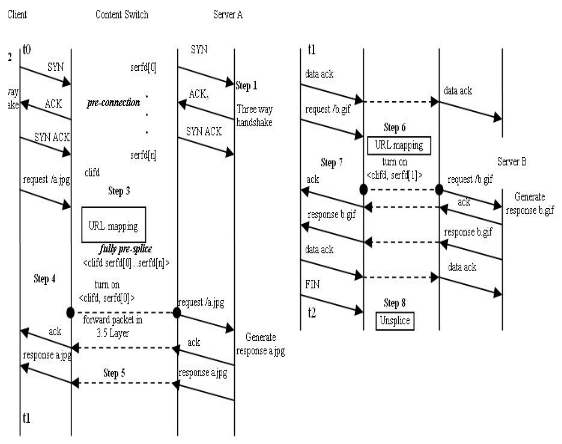


Figure 2. Time diagram of Fully Pre-Splicing.

with protocol header restored.

8) The client receives the response of the first request. The major differences between FPS and SSA appear. When this client sends several requests in the same connection, it always hits in the fully pre-splicing table. This is our point to reduce the overhead of SSA's approach. The extra overhead of our approach is caused by searching the fully pre-splicing table containing more connection entries than SSA.

B. *The Mechanism of FPS*

We present the operations of our proposed system in Figure 2. First, the content switch pre-forks a number of TCP connections to the web servers (Step1), shown as serfd[0], serfd[1], ... serfd[n] in the figure. These pre-forked connections are kept long-lived and put into the available connection list. After the TCP connection setup is completed, the client sends packets conveying the HTTP request for accessing some page (Step2). Once such a packet arrives, the dispatcher parses the URL of this request, and then looks up the URL table to select the least loaded server that possesses the requested content (Step3). Turn the connection status flag on, and send the request directly to the real serverA (Step4). When the designated server receives the request, it parses the URL to determine the requested content. It then fulfills the request and transmits the response to the client. The forwarding module also intercepts theses response packets and performs the reverse packet modification so that the client can transparently receive and recognize these packets (Step5).

When the client sends the subsequent request by HTTP 1.1's persistent connection, the forwarding module will accept and parse the request to decide the real server (Step6). If the decided server is the same as last time, the request is sent to the server with non extra-load. We get the pre-forked connection from the pre-connection list and splice clifd with serfd[1]. Of course, we should turn the clifd and serfd[1] connection flag on, and turn the last time connection flag off (Step7). If the client sends the packet with FIN flag to the real server, the content switch will intercepted the packet and close all the fully pre-spiced connections with this client (Step8).

C. *Implementation*

To implement our system and content-aware dispatching algorithms, we have modified the TCP Splicing (TCPSP project) from LVS. TCPSP implements TCP splicing mechanism within the Linux Netfilter infrastructure. The Linux Netfilter package encapsulates functional support for network address translation, firewalls, and other forms of programmable layer 4 and layer 3 filtering. The programming interface for the Netfilter package allows filtering based on IP address and port number of source and destination.

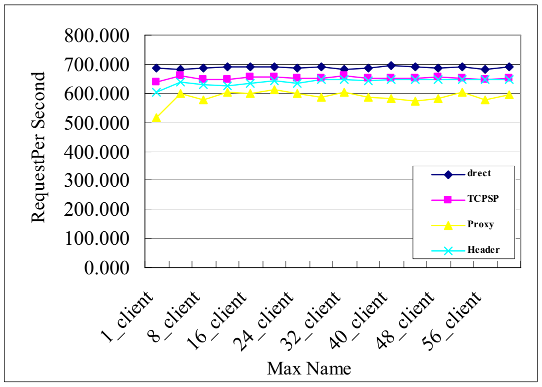
Netfilter consists of two main parts. First, each network protocol defines well defined points in a packet's traversal of that protocol stack, called "hooks". Second, kernel modules may register to listen to specific hooks of different protocols. When a packet traverses the protocol stack, Netfilter checks if any modules have registered for that protocol and hooked, in which case they are given the chance to examine, possibly alter and provide a verdict for the packet. The verdict may discard the packet, accept and allow it to continue the traversal of the protocol stack, steal it, or request Netfilter to queue it for user-space processing.

The implementation of the FPS is based on the NF_IP_LOCAL_IN hook in order to intercepts all packets that pass through the content switch. FPS intercepts the packets by the NF_IP_LOCAL_IN hook in the ip_local_deliver() function. The ip_local_deliver() function is the boundary between IP and TCP layer for the incoming path. When the packets traps into the NF_IP_LOCAL_IN hook, fps_in() function of the FPS handles the packets. FPS must check the protocol header length, checksum update, parse header, compute sequences and timestamps, dispatch and rebuild the protocol header. If the FPS finishes the works, the packet is sent to the destined computer by the ip_send() function. The ip_send() function is the boundary between IP and MAC layer for the outgoing path. Therefore, as long as the FPS handles the packets, the packets will never go to upper TCP layer.

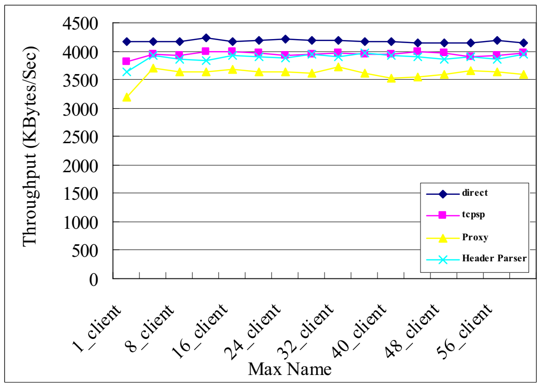
III. Performance Evaluation

Experiments are divided into three parts. The first set of experiments measured and compared the performance of four different methods, direct request (client requests server directly), application proxy, TCPSP, and TCPSP with header parsing. Header parsing parsed the header information from the client's requests and allocated a memory space for storing the information. Thus, TCPSP with header parsing meant that TCPSP parsed header information and stored it for all requests. In this experiment, we want to show the overhead of HTTP header parsing in TCPSP.

Figure 3 show a comparison of the requests per second and throughput of the TCP splice and application proxy. The test was run on the client directly requested to the web server without any content switch presented. The results of this test should represent the best possible throughput for the test software we were using. The average difference of the direct and TCP splicing is 5%, and the average difference of the direct and application proxy is 15%. We prove that the TCP splicing has the better performance than application proxy as expected.



(a) Requests per second



(b) Throughput

Figure 3: The performance result of requests per second (a) and throughput (b).

We had tested the overhead of the HTTP request header parser in TCP splicing by the header parser line of the Figure 3 and Figure 4. The overhead is less than 2% that we can accept.

The second set of experiments compares the CPU load of the cluster servers in TCPSP and FPS. We monitor the CPU load of the web server in six minutes. The client and the controller PCs were configured with Windows XP. The web switch and the web server PCs were configured with Linux RedHat 8.0. The web servers were configured with apache 2.0.4. In order to display the difference of each server CPU load, the workload in WebBench was set in HTTP 1.1 version and the HTTP requests were for dynamic cgi programs that were CPU intensive. We used the load balancing algorithm, MCRR [7], for TCPSP in the application layer and FPS in the kernel space.

Figure 4 and Figure 5 shows the results of the servers with TCP splicing in one-second intervals. Clearly, the

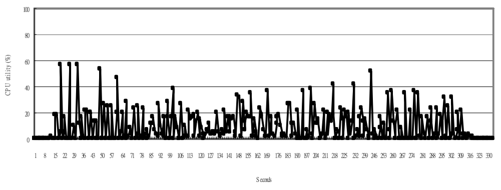


Figure 4. CPU load of the web server with TCP splicing.

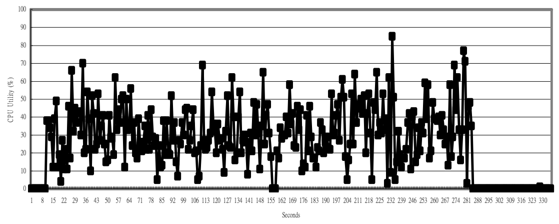


Figure 5. CPU load of the web server with FPS.

CPU load is very unbalanced in the TCP splicing. Because the TCP splicing only made the first packet of the flow with the load balancing algorithm, following packets were directly sent to the same web server. In fact, TCP splicing supported no load balancing algorithms.

As a result, we find out FPS suffers more load than TCPSP comparing others. The reason is because WebBench would not send more requests before the responses come back. The throughput results show that, TCPSP handles 10 requests per second and FPS handles 41 requests per second. So, the throughput of FPS outperforms four times than TCPSP. It is the major reason that the load in each server of FPS is higher than TCPSP. Besides, the average latency time is 202 ms in TCPSP and 39 ms in FPS, so that FPS outperforms five times than TCPSP.

IV. CONCLUSION

Fully pre-splicing mechanism (FPS) has several advantages: first, it extracts HTTP header in the kernel space to reduce the twice data copying between user and kernel space. Second, it forwards the response from server to client in the layer 3.5 of TCP/IP protocol stacks with packet header modification to decrease unnecessary data copy time from kernel to application layer. Third, it uses the pre-connection method to diminish the three-way handshake time between the switch and the server connections. Fourth, we extend the TCP splicing which is content blind to support load balancing algorithm for balancing the load of the cluster servers. Finally, FPS has succeeded reducing the overhead of SSA.

REFERENCES

[1] Maltz, D., and Bhagwat, P. "TCP Splicing for Application Layer Proxy Performance", IBM Research Report RC 21139(03/17/98) (March 1998).

[2] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. "Scalable Content-aware Request Distribution in Cluster-based Network Servers", In Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000. http://citeseer.nj.nec.com/aron00scalable.html

[3] Ariel Cohen, S. R., and Slye, H. "On the Performance of TCP Splicing for URL-aware Redirection", Second USENIX Symposium on Internet Technologies and Systems, Boulder, CO, Oct. 1999.

[4] C. Yang, M. Luo, "Efficient Support for Content-Based Routing in web Server Clusters", Proceedings of USITS'99-2nd USENIX Symposium on Internet Technologies & Systems, Boulder, Colorado, USA, October 11-14, 1999.

[5] TCP Splicing project (TCPSP), http://www.linuxvirtualserver.org/software/tcpsp/index.html.

[6] Netfilter, http://www.netfilter.org/.

[7] E. Casalicchio and M. Colajanni, "Scalable web cluster with static and dynamic contents", Proceedings of IEEE Int'l Conf. on Cluster Computing, pp. 170-177, Chemnitz, Germany, Dec. 2000