RC 22159 (99165) August 30, 2001 **Computer Science**

IBM Research Report

Evaluation of TCP Splice Benefits in Web Proxy Servers

Marcel Rosu, Daniela Rosu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Evaluation of TCP Splice Benefits in Web Proxy Servers

Marcel-Cătălin Roşu and Daniela Roşu

Abstract—**This study evaluates the performance benefits of using TCP Splice for handling non-cacheable Web objects and tunneling SSL connections in a Web proxy server. For this type of traffic, the Web proxy cannot draw any future benefits, yet it spends substantial CPU cycles moving data between server and client connections with no processing other than header manipulation. Previous studies showed the benefits of TCP Splice in reducing the overheads in proxy servers that do not need to access the content exchanged by client and server after the connection to server is established (e.g., firewalls). Our study is focused on a different type of proxy; for many of its requests, a Web proxy needs access the HTTP header sent by the server in order to assess the content cacheability. Our experimental results demonstrate that TCP Splice can benefit Web proxy servers, as well. TCP Splice enables 30-55% reductions in per-request proxy overheads, while response time is reduced by up to 25% for files larger than 10KBytes. Larger improvements are observed when handling SSL connections, in particular for small file sizes. Our evaluation is performed with a socket-level implementation of TCP Splice. Different from previously proposed TCP splice implementations, socket-level splicing allows splicing TCP connections with different characteristics. Moreover, experiments show that this implementation helps reduce the number of retransmission timeouts observed by the Web server, thus improving on the response time observed by the client. The experimental testbed includes an emulated WAN environment and benchmark applications for HTTP/1.0 Web clients, server, and proxy running on AIX machines.**

Keywords—**TCP Splice, Web proxy.**

I. Introduction

WEB proxy caches represent a popular method for improving the performance of Web serving with respect to both bandwidth consumption and user-perceived response times. Placed on the network paths that link relatively large client populations, such as university campuses and ISP's client pools, to the rest of the Internet, Web proxy caches process HTTP requests issued by clients, serving some of them with content from their local cache while the rest is served with content retrieved from the corresponding Web servers or other proxies. In addition, Web proxies tunnel SSL connections between clients and Web servers.

By serving content from their local caches, Web proxies help reduce bandwidth consumption and provide their clients with faster response times. Unfortunately, due to the current trends in Web traffic characteristics, the fraction of requests that can be serviced from a proxy cache is limited [4]. This fraction is likely to decrease in favor of non-cacheable objects and SSL connections due to emerging Web-based services; [29] reports that in proxy traces collected in 1998, 50% of the requests are for non-cacheable objects.

The performance of Web proxy servers has been addressed by an extensive body of research, most of which has focused on Web proxy components directly related to its caching functionality. For instance, previous studies have addressed topics like cache replacement policies [5], [10], cache consistency protocols [11], [13], management of memory subsystem [22], disk I/O subsystem [18]. Substantially less effort has been made to understand the performance implications of Web proxy functionality not directly related to caching, such as the networking subsystem. For instance, previous studies have addressed the implications of connection caching [8] and object bundling [28] on user-perceived response times. The work presented in this paper falls in the same category.

This study addresses the implications that handling non-cacheable Web objects and tunneling SSL connections have on the Web proxy performance and evaluates the use of TCP Splice for reducing the associated overheads. For these types of requests, the proxy moves data between server and client connections with no processing other than header manipulation. This process, also known as connection splicing, may take a significant share of the available CPU resources, without any benefits for the proxy cache.

Our study is motivated by the upward trend in the use of Web content personalization and security/privacy features which will cause an increase of the fraction of non-cacheable objects and SSL connections that Web proxies are expected to handle. As the same pool of Web proxy resources is used for serving all of the various types of Web requests that arrive at the proxy, improving the efficiency of handling non-cacheable content and SSL connections makes more resources available for serving requests for cacheable objects. The immediate effect is a reduction in proxy response times when serving cacheable objects. In

addition, for proxies that handle overload situations by forwarding requests to origin servers rather than serving them from the local cache [24], the bandwidth consumption due to overload situations is reduced.

Existing solutions for reducing the overhead of non-cacheable and SSL traffic in a Web proxy use additional network components. Placed in front of a proxy cluster, these components identify SSL traffic and some of the requests for non-cacheable content and forward them directly to origin servers, bypassing the proxy [12]. Requests are filtered based on destination port number and analysis of the HTTP header. While this solution may provide significant reductions of proxy utilization, it is not complete since content may be non cacheable even if its URL does not suggests it (e.g., the CNN home page). More important, recent studies have proved the limited scalability of content-based routing solutions [1]. With these insights and considering the downward trend in price/performance for off-the-shelf servers and network attached storage solutions, we direct our study towards single server-based solutions.

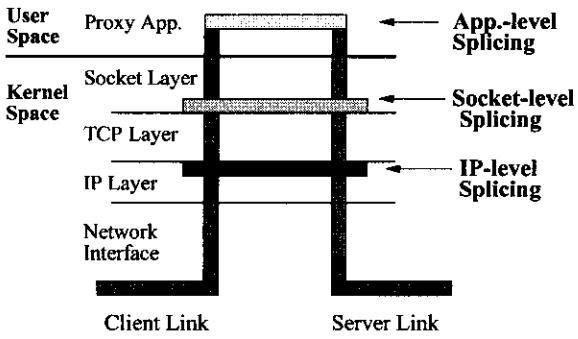

Fig. 1. Models for connection splicing.

TCP Splice kernel service was proposed in [15], [27] as a mechanism for lowering the overheads of split-connection proxies. The proposed mechanisms connect pairs of client and server TCP connections through IP forwarding and TCP sequence number rewriting. Evaluated in the context of mobile gateways [14], content-based routers [7], and TCP forwarders [27], these IP-level solutions provide 25-50% CPU load reductions.

Figure 1 illustrates the conceptual difference between application- and kernel-level connection splicing.

In this study, we use new TCP Splice mechanism implemented at the socket level. Figure 1 illustrates the conceptual differences among this socket-level splicing, the traditional, application-level splicing, and the previously proposed IP-level splicing.

Although with a higher overhead than IP-level TCP Splice, the socket-level solution isolates the characteristics of the two proxy connections and provides more flexibility to the proxy application in exploiting TCP Splice for all of the non-cacheable content that it handles. Our experimental results demonstrate that TCP Splice enables substantial reductions of proxy overheads and request latencies. TCP Splice enables 30-55% reductions in per-request proxy overheads, while response time is reduced by up to 25% for files larger than 10KBytes served over HTTP/1.0 connections. Larger improvements are observed when handling SSL connections, in particular for small file sizes. We demonstrate that additional benefits, such as faster recovery from packet loss, can be drawn from separating the packet flow on the client-proxy and proxy-server connections as enabled by socket-level TCP Splice.

For our evaluation, we use an emulated network environment. This approach allows us to study proxy behavior in a realistic and reproducible environment. For the network emulation, we use a modified version of NISTNet [20] on a PC that acts as a router among the server, proxy, and client networks.

The remainder of this paper is organized as follows. Section II discusses implications of TCP Splice in the context of Web proxies, and introduces the socket-level TCP Splice mechanism. Section III describes our experimental methodology and testbed. Section IV presents our experimental results. Section V discusses related research and Section VI concludes the paper with a summary of our results and future work.

II. TCP Splice and Web Proxies

Web proxy servers represent a particular case of split-connection proxies, servers that are interposed between the server and the client machines to mediate their communication and/or provide improved service value. A TCP connection carrying Web traffic may be split at several points between client and server, such as at the boundaries of client and server domains, or at ISP servers.

Besides the World Wide Web, sample Internet services using split-connection proxies include firewalls, gateways for mobile hosts, and stream transducers. Traditional implementations of split-connection proxy servers are based on application-level modules. Previous studies demonstrated that the performance of this type of solutions is limited because of the substantial overhead of handling a large number of active TCP connections in user space.

The *TCP Splice* kernel service was proposed in [15], [27] as a mechanism for lowering the overheads on the proxy node. This mechanism connects pairs of client and server TCP connections through IP forwarding and TCP sequence number rewriting (see Figure 1). When exploit-

ing TCP Splice, the work performed by the proxy application is significantly reduced. Namely, the application receives and processes the client request, establishes the connection to the server, sends a response to the client, and then releases the two connections to the TCP Splice kernel module which transfers data until connections are closed by the end points.

These implementations of TCP Splice result in significant reductions in CPU overheads. The proposed TCP Splice implementations can be used to handling SSL traffic and HTTP requests for content that can be categorized as non-cacheable based on parameters in the client request message. However, for other types of requests, additional mechanisms like *TCP Tap* [16] are required to enable the proxy application to retain a copy of the content passing through the TCP Splice module.

The proposed TCP Splice implementations preserve the end-to-end semantics of TCP, as the proxy node does not interfere in the flow of data and ACK packets exchanged by the endpoints, i.e., data packets are acknowledged only by the two endpoints of a data connection. This feature, deemed important for wireless base station or gateways [14], [2], comes to the expense of disabling the Web proxy application to splice connections with different TCP characteristics.

A. Socket-Level TCP Splice

To address the complex content delivery scenarios that may occur in a Web proxy, we propose and evaluate a *socket-level TCP Splice* mechanism. The fundamental difference between this mechanism and the IP-level mechanism proposed in [15], [27] is that the proxy server is the actual endpoint for its client and server connections, delivering ACKs to the server as soon as data packets are received by the proxy host, and retaining the ACKs sent by the client.

While incurring higher overheads than the IP-level implementation, the socket-level TCP Splice allows a proxy application to have direct control and full flexibility on when to start and stop the splicing. Figure 2 illustrates two ways in which the socket-level TCP Splice can be exploited by the proxy application. The two examples present the procedures for handling CONNECT requests and GET requests that require interaction with the server. The two procedures are called after the proxy accepted the client connection, analyzed the request, and opened the connection to server. When processing a CONNECT, TCP Splice is exploited in a way similar to that enabled by the IP-level TCP Splice model. However, for GET requests, the socket-level TCP Splice allows the proxy to read one or more segments of the response including the HTTP header, process it and evaluate whether the content is cacheable. Only if the content is not cacheable, TCP Splice is invoked, possibly indicating the length of the response that remains to be spliced, for HTTP/1.1 connections. For these connections, the proxy can regain the control of the two connections when the current transfer is completed and input is received on the client connection.

A socket-level TCP Splice can be integrated with mechanisms like TCP Tap to collect server content in its way to the client. An advantage over IP-level solution is that with socket-level splice, a TCP Tap-like mechanism can collect the server's reply even if the client connection goes down, enabling the proxy to minimize the negative effects of connection aborts on bandwidth consumption [8].

An important advantage of the socket-level TCP Splice is that it enables splicing between any two connections using sockets, thus precluding the limitations present in the IP-level TCP Splice regarding SACK, Maximum Segment Size, timestamps, and window scale. These features can significantly benefit proxies like those serving a large Win95 population, which lacks SACK. This advantage derives from the almost complete isolation of the two spliced connections; the extent of coupling between the two connection is determined only by their maximum data transfer rates and by the size of their send and receive buffers on the proxy.

A drawback relative to IP-level TCP Splice is that the socket-level solution violates the TCP's end-to-end reliability and correctness semantics. While this may not acceptable for some types of proxies, it is definitively tolerated by others, a Web proxy being probably the most preeminent example.

A.1 Implementation

The socket-level TCP Splice implementation evaluated in this paper is included in the 5.10 release of AIX, IBM's Unix. The basic functionality is accessible at the application level through the `splice()` system call and it is described below.

By calling `splice(A,B)`, an application relinquishes the control of the connection/socket pair to the TCP Splice. TCP Splice manages the two connections/sockets by moving data packets between A and B through simple manipulations of their receive and send buffers. After an incoming data packet on connection A is successfully processed by the TCP code, it is added to B's send buffer, if possible. The data packet is sent out on connection B as soon as the state of B's congestion control allows it. If B's send buffer is full, the packet is added to A's receive buffer; the packet is moved to B's send buffer and sent out when sufficient ACK packets are received on B.

```
ProcessCONNECT(sockClient, sockServer) {
  SendClientOK(sockClient);
  splice(sockClient, sockServer);
  close(sockClient);
  close(sockServer);
}
```

```
ProcessGET(sockClient, sockServer) {
  PrepareServerReq(dataBuffer,reqBuffer,...);
  SendReqToServer(sockServer, dataBuffer);
  read(sockServer, dataBuffer, EXPECTED_HEADER_SIZE);
  type = ProcessResponseHeader(dataBuffer);
  SendDataToClient(sockClient, dataBuffer,...);
  if( type == GET_UNCACHEABLE)
    splice(sockClient, sockServer);
  else
    ReadInCacheAndPushToClient(sockClient, sockServer,...);
}
```

Fig. 2. Sample usage of TCP Splice in handling two types of HTTP requests.

Compared to the IP-level solution [15], the socket-level implementation runs each data packet through the TCP input code of one connection and the TCP output code of the other connection. This activity incurs more CPU and memory overhead than rewriting sequence numbers in the IP-level solution. However, our experiments demonstrate that the CPU-overhead differences between the application-level splice and each of the two kernel-level splice implementations are comparable.

One of the distinguishing attributes of a socket-layer implementation is simplicity: the code consists of approximately 100 lines of C code. Although scientifically unimportant, simplicity played an important role in its acceptance by the AIX development team.

III. Experimental Environment

This section describes the environment used for the evaluations presented in the paper. Section III-A describes the hardware and software used in our experiments. Section III-B describes our experimental testbed and the emulated network conditions. Finally, Section III-C describes our experimental methodology.

A. Hardware, System Software, and Applications

Three IBM RS/6000 servers and one PC are used in our experiments. The three servers run our Web client, server, and proxy applications. The PC acts as a router that emulates wide-area network conditions. The hardware configurations of the four machines are described in Table I.

The three servers run AIX 5.10; the PC runs Redhat 7.1 Linux with the 2.4.4 kernel. The proxy applications use the `splice()` system call, which is included in the 5.10 kernel. The AIX TCP implementation is derived from BSD 4.4 and it has been previously extended with support for NewReno, SACK, Limited Transmit, ECN, and for handling large workloads [19]. On the PC, a modified version of NISTNet [20] is used for network emulation; its main component is an extension to the Linux kernel. No applications are run on the PC.

TABLE I
HARDWARE CONFIGURATION OF TESTBED.

	CPU Type	Speed (MHz)	Memory (MBytes)
Client	32-bit PPC	200	256
Server	64-bit PPC	200	256
Proxy	32-bit PPC	133	128
PC	PentiumPro	200	256

The client machine generates HTTP workload using an extended version of the `s-client` [3], which is a low-overhead, event-driven generator of HTTP requests implemented as a single-threaded process. The client machine runs one instance of the s-client, which is configured to emulate a specified number of concurrent connections/clients. For each emulated client, the s-client process loops through the following four steps: (1) open connection to proxy, (2) send HTTP/1.0 request, (3) wait for response, and (4) closes connection.

The `s-client` was extended with mechanism for generating a configurable sequence of message exchanges on a TCP connection before sending the HTTP GET command. The mechanism is used in our experiments to emulate the message exchange in the SSL handshake protocol. To emulate the load that SSL connections place on proxies, the client and server applications exchange messages of the same length and in the same order as a typical SSL handshake [25]. Note that neither client nor server machines run the SSL protocol.

The server machine runs a simple HTTP server emulator, which returns the appropriate HTTP headers and uses a random stream of characters to build the content of the response. The length of the returned document is specified by the client through an application-specific HTTP header. The server emulator implements the same mechanism for

configurable message exchange as the `s-client`.

The proxy machine runs a very simple Web proxy which understands GET and CONNECT commands and uses the `select()` system call to monitor socket activity for all of its pending connections. Upon receiving a GET command, it uses the name of the requested object to decide between serving the request locally and forwarding it to the server. The decision is based on the value of an application-specific HTTP header in the client request. When an object is to be serviced from the cache, the proxy uses the same procedure as the HTTP server emulator described above. Upon receiving the CONNECT command from the client, which signals the start of an exchange over an SSL connection, the proxy establishes a new connection to the server and starts tunneling data between the two connections. The proxy never copies data between application buffers.

The proxy can be configured to use TCP splice or to perform application-level splice. When using TCP splice for GET requests that are forwarded to the server, the proxy can splice the connection immediately after the connection to server is established or after reading the HTTP headers from the server; the actual method is selected by a configuration parameter.

B. Experimental Testbed

In all our experiments, the three servers are configured on separate subnets and the PC is acting as a router between them. The PC is also used to introduce packet losses and delays, and to enforce bandwidth limitations. For improved predictability of the emulated network, the servers were connected to the PC directly, using crossover Ethernet cables (see Figure 3). The machines are connected to the emulated network using dedicated Fast Ethernet network interfaces; additional interfaces are used to connect to the corporate network.

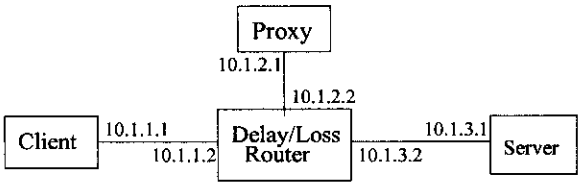

Fig. 3. Testbed configuration.

The network conditions used for the experiments described in this paper are emulated with a modified version of NISTNet. We extended NISTNet with a two-state dependent loss model in which the loss probability of a packet following a dropped packet is different from the probability of an initial loss in the packet stream. This change is motivated by several Internet studies [23], [26] which have shown that the probability of a packet being dropped is much higher when it immediately follows a dropped packet than otherwise. Another NISTNet modification is aimed at lowering the CPU overhead of network emulation: a cache of (network filter, mask)-pairs was added in front of NISTNet's packet classification code to avoid trying multiple masks for each packet. This addition complements the filter selection mechanism in NISTNet, which uses a hash-based method to select, for each packet, the set of possible filters to try. Lowering emulation overheads is particularly important as the traffic generated in our experiments traverses the PC/NISTNet machine twice.

Three wide-area network settings are used in our experiments. The first setting aims at emulating network conditions experienced by corporate or high-speed dialup users. The other settings attempt to emulate the conditions experienced by 56k-modem users. Table II presents the delay and loss parameters characterizing the emulated networks.

In addition to the above-mentioned parameters, modem or low-speed connections are limited each to 56kbits/sec downstream and 28kbits/sec upstream. These restrictions are implemented as 6000 separate NISTNet rules and the client machine is configured to allocate ephemeral TCP ports within the range covered by these rules.

Selecting the best wide-area network settings to emulate network conditions is a difficult task. Such parameters depend on a variety of factors including user connectivity, accessed Web proxy, time of day, and characteristics of the client, proxy, and server TCP implementations. Our selection of network condition parameters aims to approximate existing Internet conditions, yet represent a tradeoff between emulation accuracy and overhead.

C. Experimental Methodology

In our experiments, we vary the following factors: number of Web clients, size of transferred Web objects, type of requests, proxy configurations, and network conditions. In the experiments with homogeneous types of requests (i.e., all CONNECT or all GET), the number of Web clients varies from 20 to 140 in increments of 20. In the experiments with mixed request types, Web clients vary from 40 to 160 in increments of 40. The selected object sizes are 1, 5, 10, 15, 20, 33, 45, 64, 71, 90, 100, 110, and 128 Kbytes. With these range of parameters, the client and the server machines were never overloaded in these experiments.

The proxy can be configuration to use application- or socket-level splicing. Unless otherwise mentioned, NewReno and Limited Transmit are turned on and SACK is off.

For experiments with SSL traffic, we use message ex-

TABLE II
PARAMETERS FOR EMULATED NETWORKS: DELAY/(INITIAL LOSS, SUBSEQUENT LOSS).

	Client-Proxy	Proxy-Server
Corporate	10ms/(.1,40.)%	90ms/(1.,40.)%
Modem Loss 1%(56k)	50ms/(.1,40.)%	90ms/(1.,40.)%
Modem Loss 2%(56k)	50ms/(.1,40.)%	90ms/(2.,40.)%

TABLE III
MESSAGE EXCHANGES FOR EMULATED SSL HANDSHAKE.

Handshake	Sequence: Source(bytes)
Full	Clt(98) Srv(2239) Clt(73) Srv(6) Clt(67) Srv(61)
Abbreviated	Clt(98) Srv(79) Srv(6) Srv(61) Clt(6) Clt(61)

changes emulating both full and abbreviated SSL handshakes [25]. Among all handshakes, 60% are abbreviated [17]. The sequence of exchanged messages is described in Table III and it is based on observed traffic and TLS specification [25]. When SSL exchange is used, its overhead is included in the reported response latency.

For each experiment, we collect the statistics displayed by the client application, and the CPU utilization and TCP statistics on the proxy, server, and client machines. We use the TCP statistics to verify the emulated loss rates. Each data point is the average of six runs, where each is run is the average over sampling intervals of 120 sec. for experiments emulating corporate networks, and of $(20 \cdot file\_size/1024)$ sec for experiments emulating modem traffic. All runs have 20 sec warm-up intervals and 20 sec shutdown intervals.

All graphs showing absolute performance numbers include 90-percent confidence intervals, calculated with the T-student distribution. In graphs showing relative improvements, the improvement is computed by $(v\_app - v\_socket) * 100/v\_app$, where v_app is the value corresponding to application-level splicing and v_socket is the value corresponding to socket-level splicing; no confidence intervals are computed for these graphs.

IV. EXPERIMENTAL EVALUATION

This section presents our experimental results. Proxy latencies in forwarding request and response packets are presented in Section IV-A. Proxy overheads and end-user latencies for non-cacheable Web requests are presented in Section IV-B, and corresponding results for handling the same Web workload over SSL connections are presented in Section IV-C. The performance implications of serving requests from both proxy cache and Web server are presented in Section IV-D. The effect of using a Web proxy with socket-level TCP splice on user-perceived latencies for non-cacheable objects is presented in Section IV-E.

TABLE IV
LATENCY CHARACTERISTICS.

	App. Level	Socket Level
HTTP Req.	228,185 μs	260,686 μs
Server Resp.	1,240 μs	278 μs

A. Latency Characteristics

We measured the time it takes to the proxy machine to forward the HTTP request from client to server, and to forward the first data packet of the response from server to client. Measurements are taken for both application- and socket-level splicing. For measurements we used `tcpdump` traces taken on the proxy machine; as a result, adapter and device driver delays are not included. The client application is configured to emulate one Web client; therefore, there is only one outstanding request at any time.

The results are summarized in Table IV. As expected, the average delays in forwarding client requests are similar for the two splicing models; both delays include the $180,000\mu s$ RTT between proxy and server. Request forwarding has a bimodal distribution in both splice models and most of the two confidence intervals overlap. In contrast, the delay of forwarding server response packets is much faster for the socket-level implementation than for the application-level implementation. This is because the socket-level implementation eliminates two system calls, and the associated data copies and context switches. Although socket-level forwarding delays are larger than IP forwarding delay, which have an average of $\approx 159.8\mu s$ on the proxy machine, they provide substantial savings over application-level forwarding delays. Note that the relative savings in CPU overhead for the two TCP Splice methods are not as large as the relative difference in their forwarding delays because some of the splice-related overheads are not included, such as the interrupt overheads.

B. Handling Non-Cacheable Requests

The workload used in this section consists of Web requests for non-cacheable objects. We assume that the proxy uses the name of the requested object to decide the object's cacheability, and thus, it splices the client and server connections without waiting for the header of the server reply.

Figure 4 shows the proxy CPU overhead per request when using application-level splicing. This overhead is expressed as a percentage of the total proxy capacity. The proxy loads observed during these experiments are between 20% and 100%. Figure 5 shows the percent of CPU overhead saved by performing TCP splicing at the socket-level. We compare the percent of CPU overhead per request because socket-level splicing is characterized by somewhat higher request rates, for reasons described later in the section. For completeness, Figure 11 shows the variation of proxy CPU utilization when using application- and socket-level splicing for select number of clients.

CPU savings can be as high as 55% and depend on the number of concurrent clients and object size. For small objects, the overhead of parsing the HTTP request, establishing the connection to the server, and forwarding the request dominate. As a result, the differences between application- and socket-level splicing are smaller. CPU savings are the smallest for 1k objects because all but the data packet returned by the server are handled identically in the two splicing configurations.

For larger objects, the utilization on the proxy machine increases and the amount of data read in one operation by the application-level proxy increases. As explained next, this results in fewer ACK packets sent to the server, and lower proxy overhead. The socket-level implementation does not exhibit this intrinsic adaptability and its CPU savings decrease for large object sizes and number of clients.

ACK packets are sent by many BSD-based TCP stacks, AIX's included, either by the fast timer routine, as delayed ACKs, or after two or more data segments are read by the application. As the file size increases, we observe an increase in the ratio of the number of data to ACK packets sent/received by the server. This leads to a reduction in the load observed by the proxy. In contrast, the socket-level implementation sends an ACK packet for every two data packets received, independent of the file size and proxy load. Figure 6 shows the number of data packets sent by the server machine per ACK packet received. The socket-level splice implementation is much closer to the desired "ack every other packet" behavior than the application-level proxy. Although it increases proxy overheads, sending an ACK for every two data packets received increases the server congestion window faster and reduces the number of TCP timeout retransmissions, as shown in Figure 7.

In addition to lowering proxy overhead, socket-level TCP splicing improves the user-perceived latency and its variance. Figures 8 and 9 show the average response time of application-level splice and the reductions enabled by socket-level splicing. Figure 10 shows the relative improvements in the coefficient of variation of the latency. Note that the apparently better variation observed in experiments with application-level splice for 120 and 140 clients is due to the proxy overload condition; Figure 11 illustrates this condition. In summary, both shorter forwarding packet latencies and reduced TCP timeouts contribute to reducing response times. Part of our future work is to determine the relative contributions of these two factors.

The results in this section demonstrate that socket-level implementation TCP splice can result in CPU savings as high as 55% which are comparable to the IP-level implementation evaluated in [15]. In making this comparison, we should note that the workload used in [15] consists of long-lived connections which is different from the Web workload we used.

C. Tunneling SSL Connections

In this section, we focused on the impact of socket-level splice on proxy overheads when tunneling SSL connections. We used the same Web workload as in the previous section except for the type of client requests. In the following experiments, clients issue the CONNECT command to the proxy and run the SSL handshake protocol with the server before sending the GET command to the Web server.

Handling requests over SSL connections adds a substantial overhead to the proxy. Figure 12 shows the proxy overhead per request, as a percentage of the total proxy capacity. For small objects, using SSL connections more than doubles the proxy utilization per request (see Figure 4). The relative increase in proxy overheads decreases for larger objects.

Tunneling SSL connections with a socket-level TCP Splice reduces proxy overheads by as much as 57%, as shown in Figure 13; Figure 15 illustrates the resulting proxy CPU utilizations for select number of clients. In contrast to experiments without SSL (see Figure 5), the relative reductions in overheads are higher for smaller objects, as the SSL handshake overhead is independent of object size. Furthermore, socket-level splice reduces the latencies of secure Web requests by up to 20%, as illustrated in Figure 14.

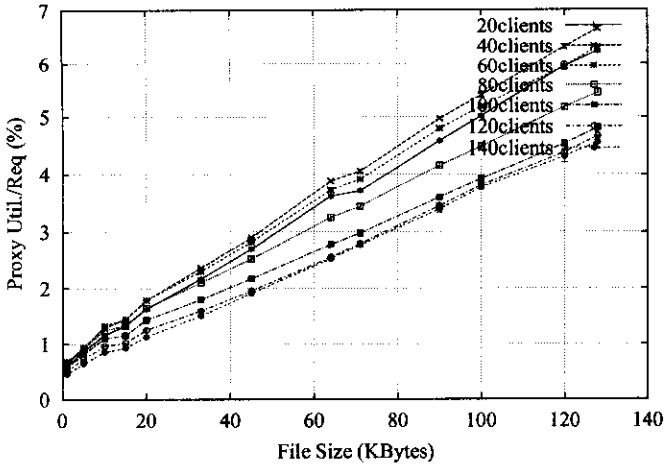

Fig. 4. Proxy Utilization per-Request for application-level splice.

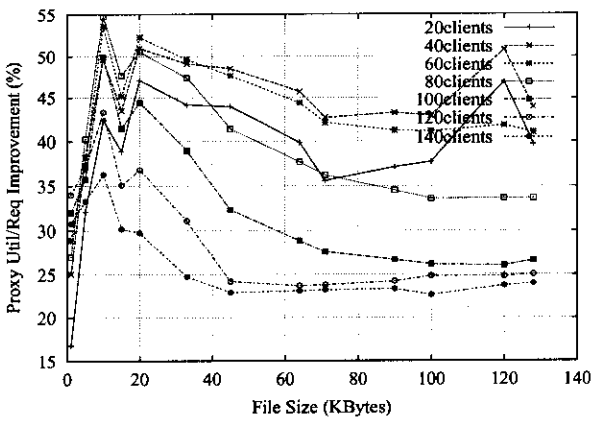

Fig. 5. Relative improvement of Proxy Utilization per-Request for socket- vs application-level splice.

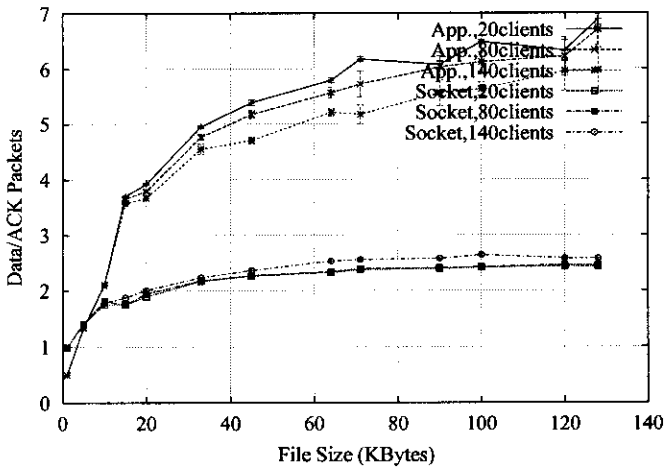

Fig. 6. Ratio of data to ACK packets received by the server.

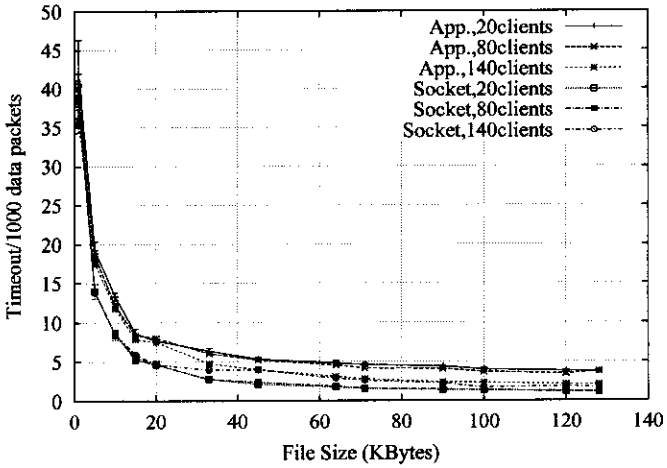

Fig. 7. Server Timeout per 1000 packets.

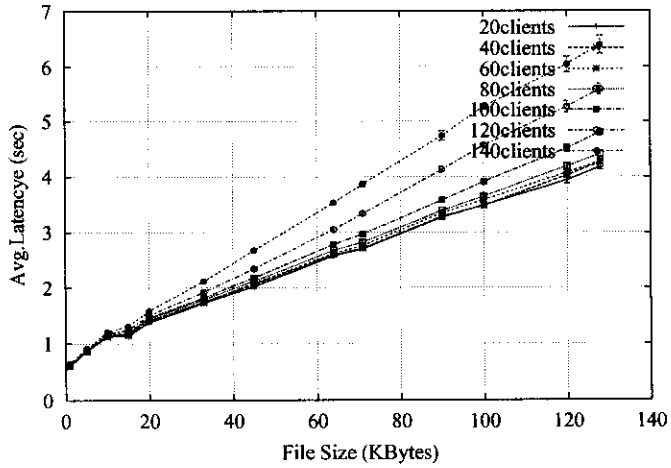

Fig. 8. Average Response Latency with application-level splice

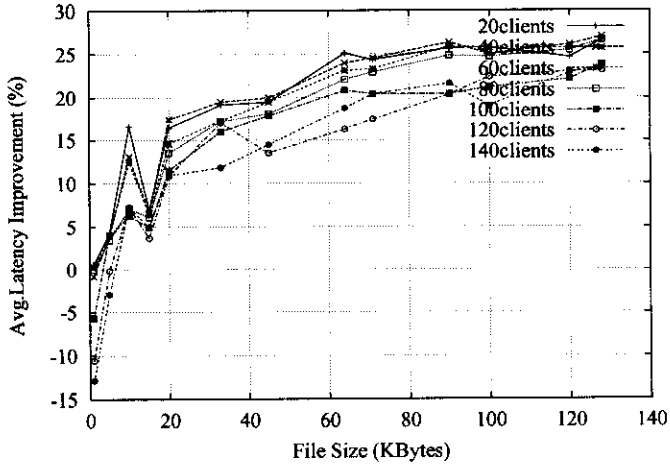

Fig. 9. Relative improvement of Average Response Latency for socket- vs. application-level splice.

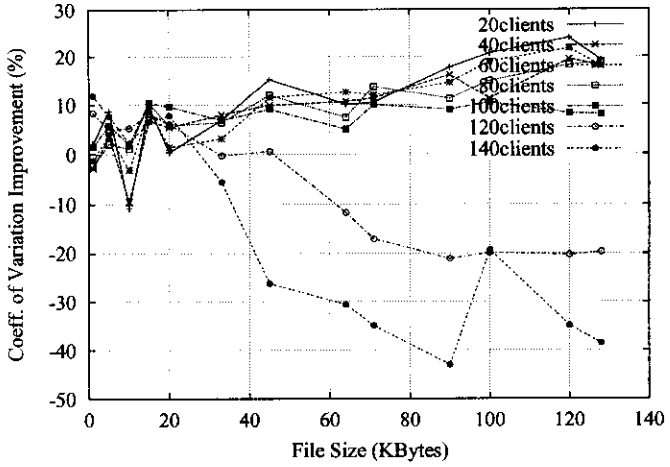

Fig. 10. Relative improvement of Latency Coefficient of Variation for socket- vs application-level splice.

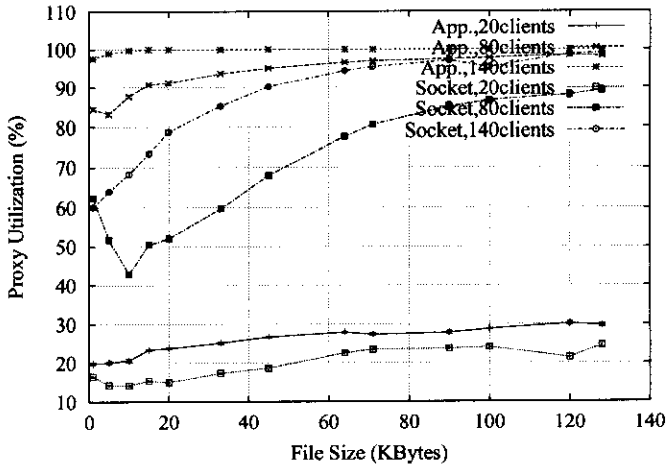

Fig. 11. Proxy utilization for socket- and application-level splice.

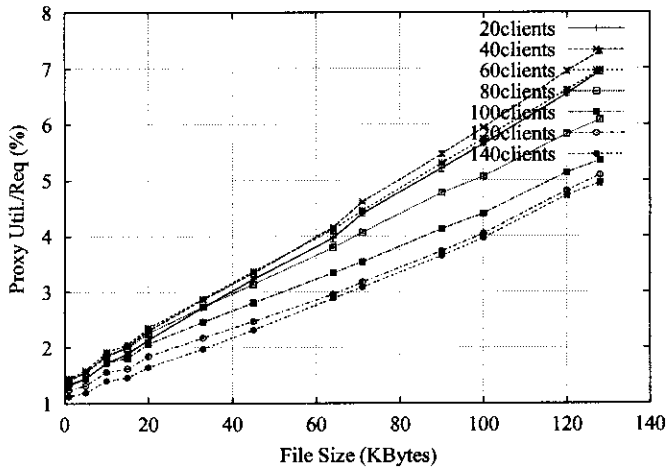

Fig. 12. SSL Traffic: Proxy Utilization per-Request for application-level splice.

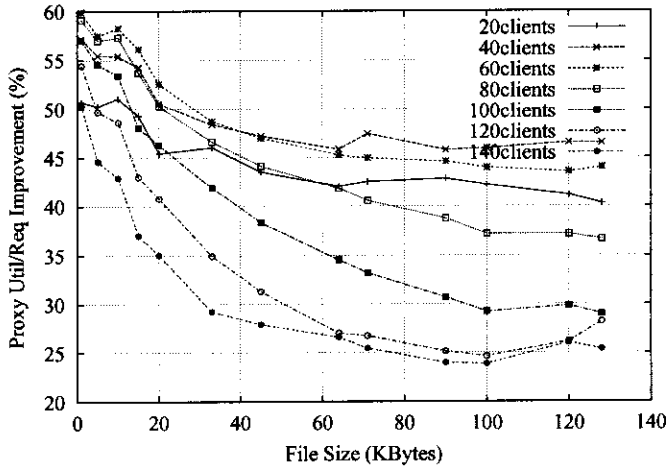

Fig. 13. SSL traffic: Relative improvement of Proxy Utilization per-Request for socket- vs.application-level splice.

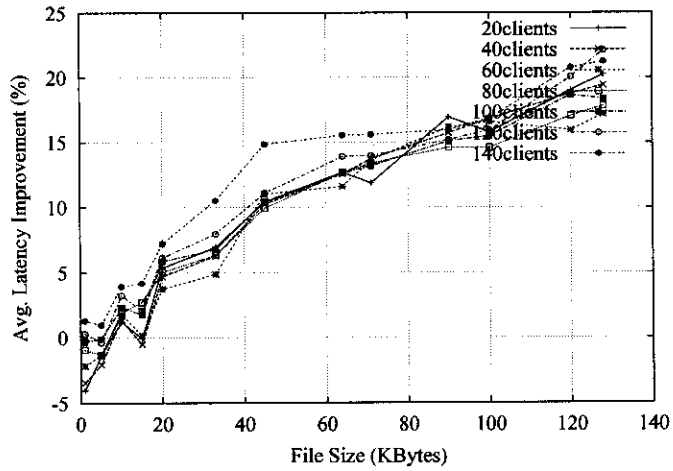

Fig. 14. SSL traffic: Relative improvement of Average Response Latency for socket- vs. application-level splice.

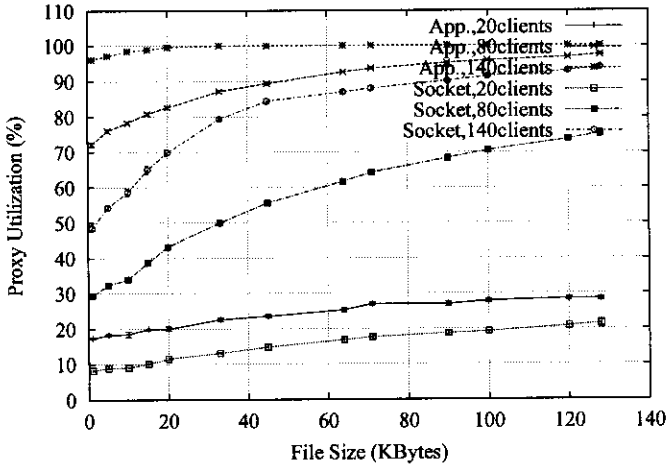

Fig. 15. SSL traffic: Proxy utilization for socket- and application-level splicers.

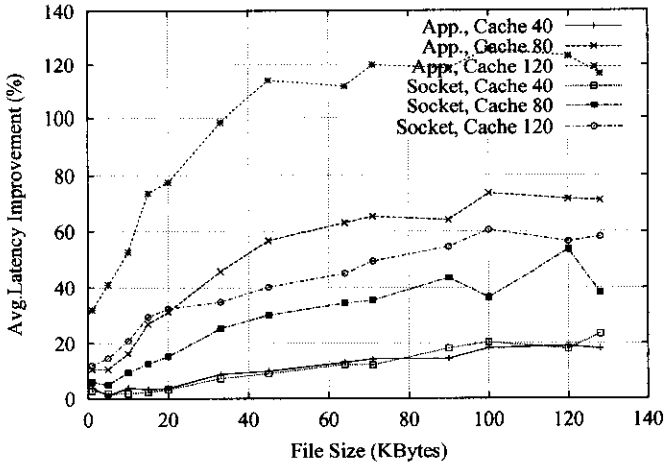

Fig. 16. Mixed traffic: Latency increase relative to latency measured for 40 clients, for SSL requests when total number of clients increases.

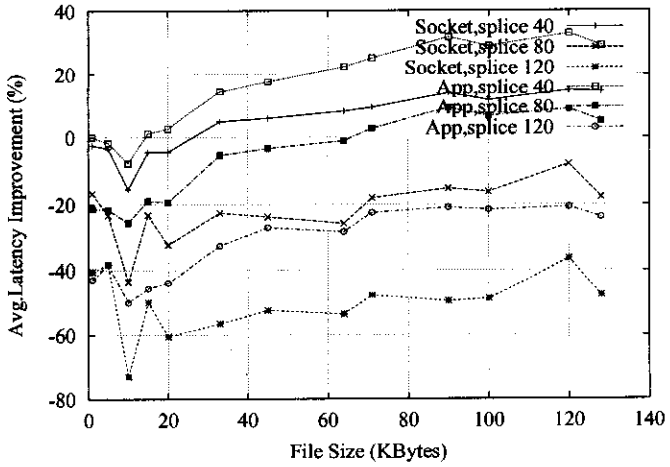

Fig. 17. Mixed traffic with increasing spliced requests and 40 cache requests: Latency improvement for cache requests relative to run with 40 cache-only requests.

D. Cachable & Non-Cachable Workload

This section presents the results of experiments with heterogeneous workloads. In these experiments, clients generate concurrent requests for cacheable objects, served by the proxy, requests for non-cacheable objects, forwarded by the proxy to the server, and requests over SSL connections, tunneled by the proxy to the server. The goal of these experiments is to observe the interactions among the different categories of requests in the context of the two TCP Splice implementations.

In the first experiment, the average number of concurrent requests that go to the server, i.e., 'spliced' requests, is fixed while the average number of requests for cacheable objects is varied between 0% to 75% of the total number of requests. Namely, out of an average of 40 spliced requests, 25 are for non-cacheable objects, 15 for SSL tunnels.

Figure 16 presents the increase of latency for tunneled connections relative to experiments with 40 clients, as the number of concurrent requests for cacheable objects is increased. The plots correspond to an average of 40, 80, and 120 concurrent requests for cacheable objects. The figure illustrates that the latency increase is substantially smaller when the requests that are forwarded or tunneled to the server are spliced at socket level rather than at application level.

In the second experiment, the number of concurrent requests that are 'spliced' by the proxy, is varied between 0% to 75% of the total number of requests while the number of requests for cacheable objects is maintained at an average of 40. No SSL connections are used in this experiment.

Figure 17 presents the relative latency improvement for cacheable objects compared to a workload composed of 40 number of requests for cacheable objects and no spliced requests. With the reference workload, the average latency increases steadily from 0.07 sec for 1 KByte objects to 0.83 sec for 128-KByte objects, with a CPU utilization above 98%. The plots illustrate that application-level splicing results in a smaller penalty on the performance of cacheable requests. The relative impact of socket-level splice increases with increase in per-request CPU utilization (i.e., file size increase), from about 2% for 1 KByte objects to about 25% for 44 KByte objects and larger.

To summarize, these experiments illustrate two facts. First, 'local' proxy activity has a smaller influence on the proxy performance as a connection splicer when TCP connections are spliced in the kernel, at the socket level. Second, kernel-level TCP splicing can induce performance penalties on the 'local' proxy activity because of the higher service priority that spliced requests observe when handled at kernel-level.

E. Low-speed Networks

The experiments described in this section compare access latencies across a slow dial-up connection in configurations with and without a Web proxy; the workload includes only requests for non-cacheable objects. As IP-level forwarding delays are much smaller than the network delays, the no-proxy experiments can be used to approximate a proxy configuration in which the proxy implements TCP splice at the IP-level [15]. The difference between access latencies in the IP-level proxy configuration and the no-proxy configuration can be considered constant and equal to the IP-level forwarding delay.

For the Web proxy experiments, we use the network conditions described in Table II and present the results of two sets of experiments: one with SACK disabled and one

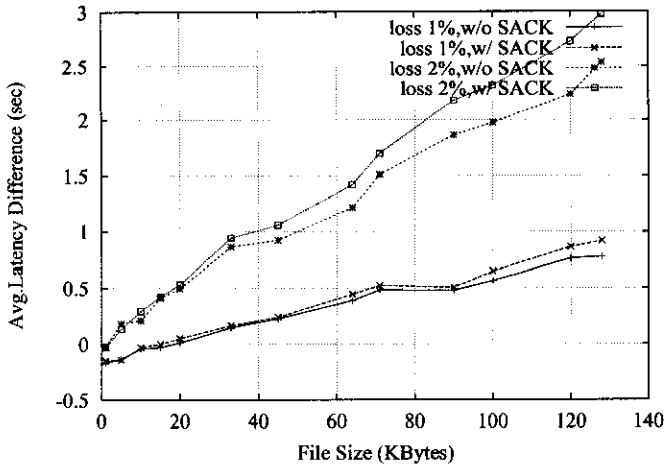

Fig. 18. Difference in latency without proxy and with proxy using socket-level splice, for 20 clients.

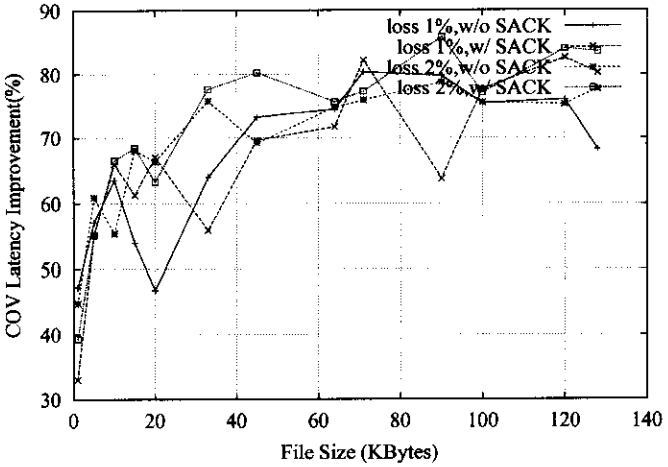

Fig. 20. Improvement in latency COV with proxy using socket-level splice vs. no proxy, for 20 clients.

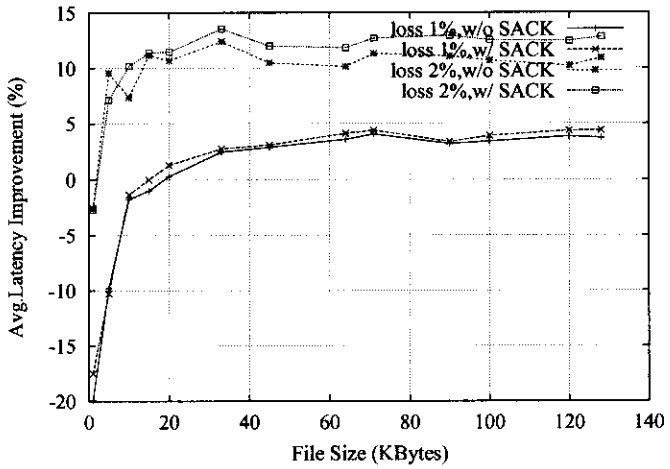

Fig. 19. Latency decrease with proxy using socket-level splice vs. no proxy, for 20 clients.

with SACK enabled on the server and proxy machines. SACK was always disabled on the client machine. For the no-proxy experiments, the network latency and initial loss rate are set to the sum of the corresponding parameter for the networks between client and proxy, and between proxy and server; the conditional loss rate and bandwidth limit are the same as in the proxy experiments. The initial loss rate between proxy and server is set to 1% and 2%.

Figure 18 shows the difference in user-response times between the no-proxy and proxy configurations in experiments with 20 clients. For small files and lower loss rate, the latency difference is negative and dominated by the request forwarding delay. For larger files or larger loss rate, the difference becomes positive as the higher ACK rates generated by the proxy (see Figure 6) help reduce average transfer times and reduce their variance (see Figure 20). Note that, for all file sizes, the minimum response times in the no-proxy configuration are $\approx 200ms$ lower than the corresponding times in the Web proxy configuration. However, both maximum response times and standard deviations are much larger in the no-proxy configuration.

It is worth noting that the observed latency differences which are up to 0.75 sec. for 1% loss and 2.5 sec for 2% loss, do not translate in significant relative improvements (see Figure 19). This behavior is because, in the emulated networks, transfer times get significantly higher as file size increases.

Our experiments also illustrate that using SACK between server and proxy can further improve the average response times. This benefit is independent on whether the client TCP stack implements the SACK feature. Moreover, this benefit is specific to the socket-level TCP splice proposed in this paper, as IP-level implementations of TCP splice cannot improve the TCP transfer rates between client and server machines.

V. Related Work

Recent research on Web server performance has focused on optimizing the operating system functions on the critical path of request processing. For instance, [22] proposes a unified I/O buffering and caching system that eliminates all data copies involved in serving client requests and eliminates multiple buffering of I/O data. In this paper, we focus on the overheads of the I/O subsystem, proposing a mechanism that Web proxies can use in a flexible manner to handle non-cacheable content and SSL tunneling.

Numerous studies on TCP and server performance demonstrate that the overhead of copying data between kernel and user-space buffers is a limiting factor for the achievable transfer bandwidths. For instance, [6] demonstrate about 26% bandwidth improvement at 8KB MTU,

when the application is not touching the data, like a Web proxy handling SSL connections and non-cacheable content. The socket-level TCP Splice mechanism proposed in this paper is particularly targeted to reduce these overheads, experimental results demonstrating CPU overhead improvements of about 55%.

Previous research has proposed and evaluated several mechanisms for kernel-level splicing of TCP connections. Some of these solutions [21], [9], [7] do not make the splicing interface available at application level. These solutions are integrated with kernel-level modules for HTTP request distribution and are implemented either between the TCP and socket layer [21] or at IP-level [9], [7] The evaluation in [7] compares content-based router implementations based on kernel and application-level splice, demonstrating that kernel-level TCP Splice benefits the performance for both short- and long-transfer connections. Extending these results, our research evaluates the impact of TCP Splice on the performance of all of the nodes participating in the transfers and considering different levels of proxy load, request types, and network conditions.

Kernel-level TCP Splice mechanisms that can be exploited at application level have been proposed in [15], [27]. The proposal in [15] is based on IP forwarding. Aiming to preserve the end-to-end semantics of the TCP model, this splicing model does not interfere with the flow of ACK packets sent by the two endpoints and require that the spliced TCP connections have similar characteristics. The proposal in [27], implemented in Scout, a communication-oriented operating system, is based on TCP-level forwarding and can be optimized to work at IP-level like [15]. Similar to [15], the splicer does not interfere with the ACK flows and require identical TCP connection characteristics.

By interfering with the ACK flows, the socket-level TCP Splicing method proposed in this paper enables the splicing of connections with different characteristics, which might benefit many Web proxy deployments. As illustrate by our experimental results, this feature reduces the number of retransmission timeouts, improving on the client-perceived response times. Despite the additional TCP packet processing, the socket-level TCP splicing enables reductions in CPU overhead comparable to those achievable with IP-level splicing. Another contribution of our study is the actual evaluation of TCP splicing benefits in the context of typical Web proxy workloads, including regular HTTP and HTTP over SSL traffic.

VI. Conclusion

Web proxies spend substantial CPU cycles for handling non-cacheable Web objects and tunneling SSL connections, traffic for which the proxy cannot draw any future benefits. This paper proposes and evaluates a socket-level implementation of TCP Splice, which is characterized by overhead savings comparable to those enabled by IP-level implementations. In addition, this implementation does not limit the set of TCP features that can be used on the two spliced connections to the subset of features supported by both endpoints.

Experiments conducted across an emulated wide-area network show that a socket-level implementation of TCP Splice reduces the proxy overheads associated with handling non-cacheable objects by 30-50%. The reductions in the overheads associated with tunneling SSL connections are higher. Our experiments show that, in certain network conditions, a Web proxy can improve response times not only when serving objects from the proxy cache, but also when proxying objects from remote servers.

To conclude, we submit that Web proxies should use socket-level implementations of TCP Splice, which can significantly benefit transfer rates for any combinations of client and server characteristics while introducing request-forwarding delays similar to other types of TCP Splice implementations. In the future, we would like to investigate the effect of proxy acking policies on proxy overhead and on user-perceived response times.

Acknowledgements

Authors would like to acknowledge the major contribution of Venkat Venkatsubra to the implementation of the socket-level TCP Splice in AIX 5.1. Authors thank Suresh Chari for his help in understanding SSL message patterns.

References

[1] M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel, *Scalable Content-aware Request Distribution in Cluster-based Network Servers*, Proceedings of Annual Usenix Technical Conference, 2000.

[2] H. Balakrishnan, S. Seshan, E. Amir, R. Katz, *Improving TCP/IP Performance over Wireless Networks*, ACM International Conference on Mobile Computing and Networking (Mobicom), 1995.

[3] G. Banga, P. Druschel, *Measuring the capacity of a Web server under realistic loads*, World Wide Web Journal, 2(1), May 1999.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, *Web Caching and Zipf-like Distributions: Evidence and Implications*, INFOCOM, 1999.

[5] P. Cao, S. Irani, *Cost-Aware WWW Proxy Caching Algorithms*, USENIX Symposium on Internet Technologies and Systems, 1997.

[6] J. Chase, A. Gallatin, K. Yocum, *End-System Optimizations for High-Speed TCP*, IEEE Communications, 39(4), Apr. 2001.

[7] A. Cohen, S. Rangarajan, H. Slye, *On the Performance of TCP Splicing for URL-aware Redirection*, USENIX Symposium on Internet Technologies and Systems, 1999.

[8] A. Feldmann, R. Caceres, F. Douglis, G. Glass, M. Rabinovich, *Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments*, IEEE INFOCOM, 1999.

[9] G. Hunt, G. Goldszmidt, R. King, R. Mukherjee, *Network Dispatcher: A Connection Router for Scalable Internet Services*, International World Wide Web Conference, 1998.

[10] S. Jin, A. Bestavros, *Popularity-Aware Greedy Dual-Size Web Proxy Caching Algorithm*, International Conference on Distributed Computing Systems, 2000.

[11] J. Yin, L. Alvisi, M. Dahlin, A. Iyengar, *Engineering server-driven consistency for large scale dynamic web services*, International World Wide Web Conference, 2001.

[12] E. Johnson, *Increasing the Performance of Transparent Caching with Content-Aware Cache Bypass*, International Web Caching Workshop, 1999.

[13] C. Liu, P. Cao, *Maintaining Strong Consistency in the World Wode Web*, International Conference on Distributed Computing Systems, 1997.

[14] D. Maltz, P. Bhagwat, *MSOCKS: An Architecture for Transport Layer Mobility*, INFOCOM, 1998.

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

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

[17] D. Menasce, V. Almeida, *Scaling for e-Bussiness*, Prentice Hall, 2000.

[18] E. Markatos, M. Katevenis, D. Pnevmatikatos, M. Flouris, *Secondary Storage Management for Web Proxies*, USENIX Symposium on Internet Technologies and Systems (USITS), 1999.

[19] E. Nahum, M. Roşu, S. Seshan, J. Almeida, *The Effects of Wide Area Conditions on WWW Server Performance*, SIGMETRICS, 2001.

[20] National Institute of Standards and Technology, *NIST Net Home Page*, http://snad.ncsl.nist.gov/itg/nistnet.

[21] IBM Corporation, *IBM Netfinity Web Server Accelerator V2.0*, http://www.pc.ibm.com/ us/solutions/ netfinity/ server_accelerator.html.

[22] V. Pai, P. Druschel, W. Zwaenepoel, *IO-Lite: A Unified I/O Buffering and Caching System*, USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1999.

[23] V. Paxon, *End-to-end Internet packet dynamics*, IEEE/ACM Transactions on Networking, 7(3), June 1999.

[24] Web Polygraph, *Data Communications Tests, July 1999*, http://polygraph.ircache.net/Results/dcomm-1.

[25] T.Dierks, C. Allen, *The TLS Protocol, Version 1.0*, IETF, Network Working Group, RFC 2246.

[26] D. Rubenstein, J. Kurose, D. Towsley, *Detecting shared congestion of flows via end-to-end measurement*, SIGMETRICS, 2000.

[27] O. Spatscheck, J. Hansen, J. Hartman, L. Peterson, *Optimizing TCP Forwarder Performance*, IEEE/ACM Transactions on Networking, 8(2), April 2000, also Dept. of CS, Univ. of Arizona, TR 98-01, Feb.1998.

[28] C. Wills, M. Mikhailov, H. Shang, *N for the Price of 1: Bundling Web Objects for More Efficient Content Delivery*, International World Wide Web Conference, 2001.

[29] W.Alec, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, *On the scale and performance of cooperative Web proxy caching*, ACM Symposium on Operating Systems Principles, 1999.