Improving HTTP Caching Proxy Performance with TCP Tap

David A. Maltz
Dept. of Computer Science
Carnegie Mellon University
dmaltz@cs.cmu.edu

Pravin Bhagwat
IBM T.J. Watson Research Center
pravin@watson.ibm.com

Abstract

Application layer proxies are an extremely popular method for adding new services to existing network applications. They provide backwards compatibility, centralized administration, and the convenience of the application layer programming environment. Since proxies serve multiple clients at the same time, they are traffic concentrators that often become performance bottlenecks during peak load periods. In this paper we present an extension of the *TCP Splice* technique [6] called *TCP Tap* that promises to dramatically improve the performance of an HTTP caching proxy, just as TCP Splice doubled the throughput of an application layer firewall proxy.

Keywords: TCP, HTTP Caches, Application Layer Proxies, Performance

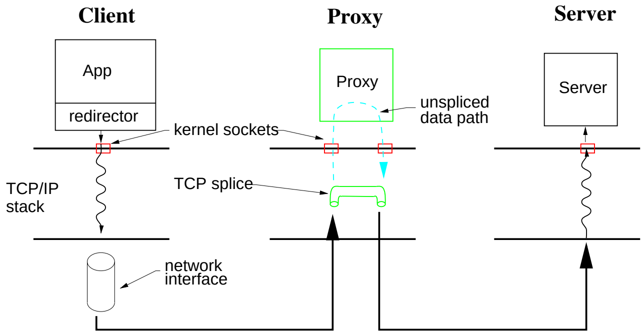

Figure 1 A generic Application Layer Proxy system showing the position of a TCP Splice.

1 Introduction

Many designs for Internet services use *application layer split-connection proxies* in which a proxy machine is interposed between the server and the client machines in order to mediate the communication between them. Split-connection proxies have been used for everything from HTTP caches to security firewalls to encryption servers[11]. Split-connection proxy designs are attractive because they are backwards compatible with existing servers, allow administration of the service at a single point (the proxy), and typically are easy to integrate with existing applications.

While attractive in design, modern application layer split-connection proxies often suffer from three related problems: they have poor performance; they add a significant latency to the client-server communication path; and they potentially violate the end-to-end semantics of the transport protocol in use. The performance and latency problems both stem from the application layer nature of the proxies. The semantics problem stems from their split-connection nature. TCP Splice [6] is a new technique that solves these three problems for some classes of split-connection proxies, such as firewall proxies, which spend most of the processing resources at the proxy moving data between the two connections. The technique involves pushing data movement from the application to the transport layer, thereby saving significant copying and processing overhead, and performing suitable address and sequence number mapping on the TCP and IP packet headers, thereby causing the two split-connections to have the exact semantics of a single end-to-end connection. When using TCP Splice, the proxy application is only responsible for setting up connections; data movement occurs at the TCP layer and is completely hidden from the proxy application.

This paper explains a new technique, called *TCP Tap*, for collecting a copy of the data forwarded over a TCP Splice and making it available to the proxy application. TCP Tap extends TCP Splice to support the class of proxies which need to read the data that flows through them, such as HTTP [1, 4] and PICS proxies [8]. Using a combination of TCP Splice and TCP Tap, we describe how it is possible to build HTTP caching proxies that can provide better throughput in addition to reducing access latency for all web clients.

Before explaining the new TCP Tap technique, we begin by briefly describing the TCP Splice technique it is based on and the performance improvements TCP Splice has brought to some application layer proxies.

2 TCP Splice

TCP Splice is a technique with the unique ability to splice together two TCP connections that were independently set up with a proxy and which may have already carried arbitrary traffic between the end-systems and the proxy application. While the two TCP connections can be used independently before the splice is set up, after the splice is created it appears to the endpoints of the two connections (client-to-proxy and proxy-to-server) that those two connections are, in fact, one. This property makes TCP Splice ideally suited for use in application layer proxy systems, since it dovetails well with existing protocols. Either end-system can exchange control information with the proxy application over its

TCP connection, and the proxy application can then step out of the communication path when its services are no longer needed.

The intuition behind TCP Splice is that we can change the headers of incoming packets as they are received at the proxy and then immediately forward the packets. In conventional proxy systems, received packets are passed up through the protocol stack to application space, where they are then passed back down again in order to be sent out. A proxy using TCP Splice, on the other hand, effectively turns the proxy machine into a layer 4 router, as shown in Figure 1. Authentication, logging, and other control tasks are done by the proxy in application space as normal, but the data copying part of the proxy — where the performance is normally lost — is replaced by a single kernel call to set up the splice. After the splice is initiated, the application layer control code can go on to other tasks.

There have been earlier proposals for relaying data between two connections inside the kernel, but TCP Splice achieves a significantly tighter binding between the connections with a corresponding savings in resources needed at the proxy. In the other proposals, the two connections that make up the logical client-server connection have a normal, complete protocol state machine running the endpoint of the connections at the proxy. The only way in which the two connections are related is that the input buffer, which normally holds data received from one connection and waiting to be read by the proxy application, is used as the output buffer for the other connection. By moving received data from the input buffer directly to the output buffer, the systems save the overhead of copying the data through application space. In TCP Splice, there *is no* input or output buffer. Received data packets are altered and then immediately forwarded. There is no protocol state machine at the "endpoints" on the proxy. There are no buffers or timers to manage, and the proxy does not send retransmissions, as happens in the other systems.

2.1 TCP Splice Implementation

A detailed description of how to implement TCP Splice is presented in [6]. This section gives a very high-level overview in order to explain the performance and semantics benefits of TCP Splice.

A TCP Splice between two connections is accomplished by altering all the packets received on one connection, including the acknowledgments, so that the packets appear to belong to the second connection, and then sending the packets out over the second connection. Since the alterations are a simple mapping function and require no storage, they can be done quickly in the kernel. Since the TCP Splice code itself does not generate data acknowledgments, TCP end-to-end reliability semantics are preserved between the two endpoints. Conventional proxy systems, on the other hand, can violate TCP reliability semantics because data sent by one end-system is acknowledged by the proxy as soon as the proxy receives it. If the acknowledged data cannot be delivered to the other end-system for any reason, that data is silently lost.

When two connections are spliced together, the data sent to the proxy on one connection must be relayed to the other connection so that it appears to seamlessly follow the data that came before it. The seamless nature of the data must be preserved even if there are data or ACK packets in flight at the time the splice is initiated, or if data must be retransmitted. Since all data bytes in TCP are assigned a sequence number in the sequence space of their connection, we achieve a seamless splice by mapping the sequence numbers from one connection's sequence space to the other connection's sequence space.

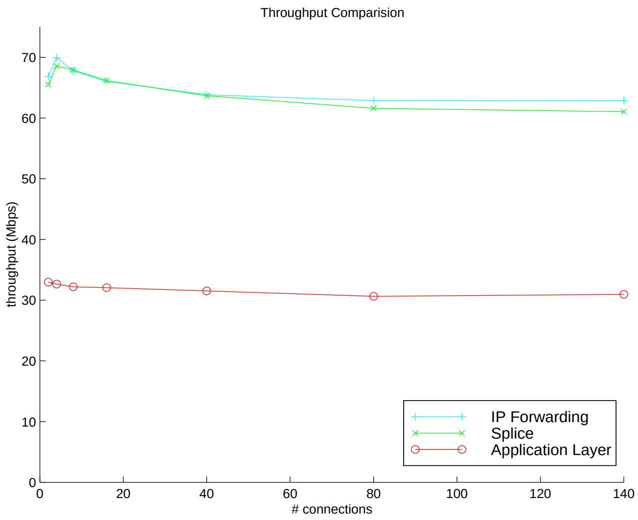

Figure 2 Throughput in Mbps of the application layer and TCP Splice proxies compared against IP Forwarding throughput.

2.2 TCP Splice Performance

In our lab experiments, we have found that a TCP Splice proxy can outperform an application layer proxy by more than a factor of two in throughput tests. For example, Figure 2 shows the results of three tests conducted a client, proxy, and server (all running BSDI Unix 3.0) connected by two separate 100 Mbps Ethernets. The proxy is a 166Mhz Pentium machine while the client and server are both running on 200Mhz Pentium pro processors. In the first experiment, the client opens multiple connections through an application layer proxy, and then pushes data through them as rapidly as possible. The server accepts connections from the client through the proxy, and then reads data from the connections. The throughput of the proxy is measured as the number of bytes per second read by the server, summed over all connections, since all of those bytes of data had to flow through the proxy. In the second test, we replaced the application layer proxy with a proxy using TCP Splice. In the final test, we configured the machine on which the proxy had run as an IP router (now performing no proxy functions) and measured the throughput achieved by its IP Forwarding loop. Figure 2 shows the results of these tests. The TCP Splice proxy supports substantially higher throughput than an application layer proxy and can sustain throughput comparable to the same hardware acting as a router.

Figure 3 compares the CPU utilization required to support a throughput of approximately 33Mbps in the three configurations. This is the largest throughput the application layer proxy could support, clearly showing that the TCP Splice proxy uses fewer

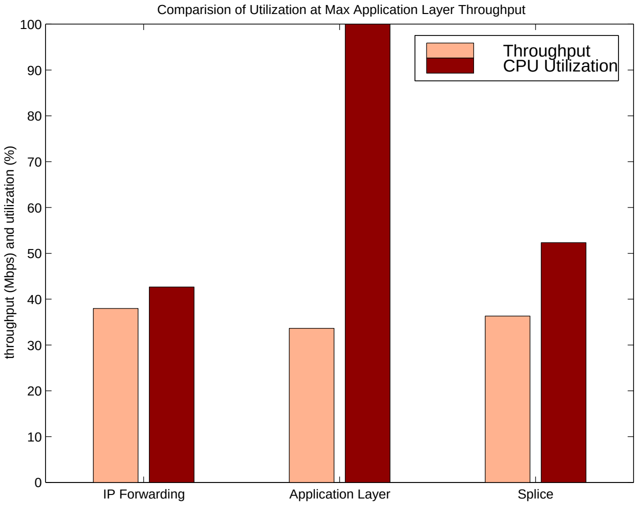

Figure 3 CPU utilization for approximately equal throughputs using the same machine as either an IP router, an application layer proxy, or a TCP Splice proxy.

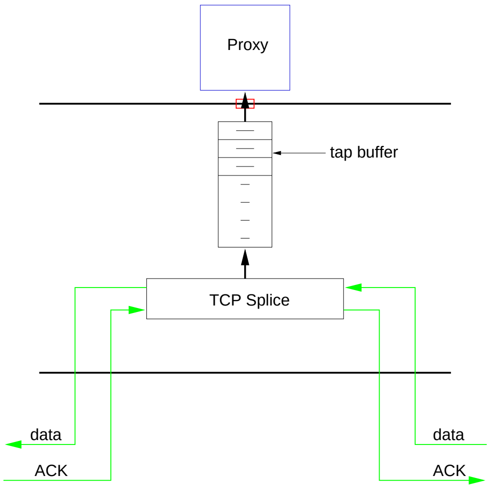

Figure 4 A TCP Tap captures the data flowing from right to left out of a spliced connection in the tap buffer, which can be read by the proxy application process via a tap socket.

proxy machine resources to do to the same work as an application layer proxy.

The traffic load for some proxies (like many HTTP caching proxies) consists of many short connections, so the proxies' ability to handle many connections per second is crucial. Preliminary results show TCP Splice proxies can support substantially more connections per second than conventional proxies, due to the very low latency of TCP Splice set-up and the way TCP Splice reduces a proxy's demand for socket and buffer resources.

3 TCP Tap and its Applications

Given the dramatic improvement TCP Splice gave to one type of application layer proxy, we would like to extend the technique to other classes of proxies. An HTTP proxy could benefit from the

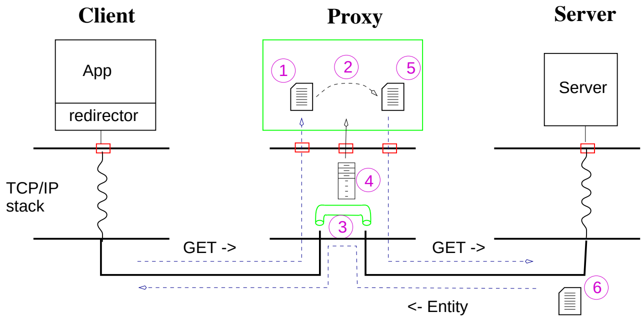

Figure 5 The steps taken to process a HTTP GET request by a proxy using TCP Tap.

TCP Splice technique if provided with a method for obtaining a copy of the fetched document to add to its cache. To obtain a copy of the fetched document, the proxy can insert a tee junction into the splice: as packets flow through the proxy, they are relayed as in a normal TCP Splice, and a copy of the data is also kept in a local socket buffer reassembly queue, called the splice's *tap buffer*, which the proxy application can read. This use of TCP Tap is illustrated in Figure 4.

The TCP Tap can be easily integrated into an application layer HTTP/1.0 caching proxy [1] as follows: Web browsers using a caching proxy first connect to the proxy and transmit their HTTP GET request. If the cache already holds the requested entity and the contents of the cache are valid, the application layer proxy serves the entity from its cache. When the requested entity is not found in the cache, the proxy application connects to the server, sends the client's GET request over the server connection, and then splices the client-proxy and proxy-server connections together. As the requested data flows through the splice from server to client, the proxy application reads the data from the tap and spools it into the proxy's cache.

Figure 5 shows all steps a proxy using TCP Tap in our design would take to process a cache miss.

1. The proxy receives a GET from the client.
2. The proxy consults its cache, and if it must fetch the requested entity from the server it converts the GET headers into the form that can be sent to the server as is standard practice for HTTP caches [1, 4].
3. The proxy opens a connection to the server, and splices the client-proxy connection to the proxy-server connection. The proxy tells the splice that it intends to write the number of bytes in the GET header into the proxy-server connection before the splice completes.
4. The proxy requests a tap on all data flowing from the server to the client through the splice.
5. The proxy sends the modified GET request to the server.
6. When the server replies with the document, the document is directly relayed to the client, and a copy is made available to the proxy application through the tap.

If the requested entity is found in the cache, but is no longer valid (as indicated by the expiration time of the cached object), the proxy sends a conditional If-Modified-Since GET request to the origin server. The proxy must defer TCP Splice establishment until after the response header is received from the origin server, because at the time it forwards the GET request it does not know whether the requested entity will be provided by the server or the cache of the proxy itself. If the origin server returns 304 Not-Modified status code, meaning the contents of the cache are still valid, the proxy serves the requested entity from its cache as before. On the other hand, if the requested entity has been modified, the origin server supplies the updated entity in the reply message. The HTTP proxy parses the reply header, writes a suitable response header into the client socket and then splices the two connections.

Since the server writes the body of the requested entity into the connection to the proxy immediately after the response header, the proxy application must tell the TCP Splice code where the break lays between data that the proxy application wants to read, and the data that should be spliced directly to the client. The proxy application indicates that it has read all it needs to from its server connection by setting a new socket option flag called QUEUE_DATA on the socket. All data received from the server after the flag is set is not acknowledged to the server, but is queued for delivery to the client as soon as the splice completes. Some data may arrive and be acknowledged before the QUEUE_DATA flag is set, and the proxy assumes responsibility for the reliable transmission of that data. It does this by telling the splice that it intends to write those bytes into the proxy-client connection before the splice completes. Queued packets are forwarded using the normal splice forwarding logic thereafter.

The HTTP 1.1 protocol [4] introduces several new complexities to the role of the proxy, and especially a proxy using TCP Splice and TCP Tap. In HTTP 1.1, the client makes one connection to the proxy, and then sends the proxy multiple HTTP GET requests over that one connection. We plan to support HTTP 1.1 caching proxies using a combination of TCP Tap and the TCP Resplice operation [5], which allows sets of arbitrary TCP connections to be spliced together, unspliced, and spliced back together again. Since some of the requested documents may lay in the proxy cache while others must be fetched from the server, the proxy uses a special TCP Tap to terminate the connection of data flowing from client to proxy and listen to the HTTP GET requests. If the requested document is not in the cache, the proxy sends the GET request over the connection to the server, fetching the document and using a tap to cache its contents as described above. If the requested document is in the cache, the proxy unsplices the server-to-proxy data connection from the proxy-to-client data connection and splices into the proxy-to-client connection the cached data. Once the cached data is transmitted, the proxy resplices the server-proxy connection back into place.

4 TCP Tap Implementation

This section describes the design of our TCP Tap implementation, though we are still in the process of implementing and testing it.

A TCP Tap can only be inserted into an existing splice, and taps the data flowing in only one direction through the splice (e.g., client to proxy to server). If the proxy application wishes copies of the data flowing in both directions, it must create two TCP Taps: one for each direction.

When a TCP Tap is created on a spliced connection, a TCP reassembly buffer is set-up and attached to the spliced connection for data flowing in the specified direction. The proxy application reads data out of the TCP Tap via a read-only socket. The tap socket is similar to a normal TCP socket, though certain socket options (such as sending keep-alives) have no effect, and there are no send buffers or timers associated with the socket. As a result, maintaining a TCP Tap into a spliced connection is a relatively inexpensive operation.

The basic operation of a TCP Tap is identical to that of the normal TCP reassembly buffer [9]. As data flows through the splice, data not currently in the reassembly buffer are added in the appropriate place. However, as described earlier, there is no TCP state machine associated with the tap buffer at the proxy. Although the proxy has no way to request retransmission of any data packets that are dropped somewhere between the server and the client, the tap buffer is still guaranteed to see all the data that flows through the logical connection. If a packet is dropped after it passes through the proxy, the tap buffer will have a copy of the data and will ignore the retransmission eventually made by the packet's sender. If a packet is dropped before it passes through the proxy, a hole is left in the tap buffer that will eventually be filled when the final receiver requests a retransmission or the sender times out and retransmits.

The key difference between a tap buffer and a TCP reassembly buffer is their handling of buffer overflow. TCP reassembly buffers can not overflow, since they are directly flow controlled and the other end-system is told not to send data that will not fit in the reassembly buffer. Tap buffers, on the other hand, are passively listening to TCP traffic that flows by them. They are not part of the end-to-end flow control, so they may receive more data at once than they can handle, nor can they request retransmission of data that overflowed, since that would violate the end-to-end semantics of the spliced connection.

If a TCP Tap buffer receives data that falls beyond the end of the buffer, the buffer drops data from the front of the buffer in order to make room to hold the newest data. The number of bytes dropped from the front of the buffer are added to the tap buffer's *gap counter*, which records how many dropped bytes lay between the last byte of data read by the proxy application, and the byte of data currently at the head of the buffer. Whenever the gap counter is non-zero, the proxy application can tell that the tap buffer overflowed and data has been dropped. The tap socket supports one additional socket operation, `TAP_GAP_COUNT`, which allows the proxy to read the current value of the gap counter.

So long as the application layer proxy process can read data from the tap buffer at the average rate it arrives, with very little overhead the proxy can obtain a copy of the fetched document. If the splice Tap buffer overflows, however, the proxy must decide whether to close the tap buffer and abandon its attempt to keep a copy of the fetched document or to somehow make use of the partial document. There are several policies a proxy could use to control how it manages tapped connections.

- The proxy can vary the size of the tap buffers it creates as system resources fluctuate.
- The proxy can use a dynamic algorithm in which if data arrives too fast for the proxy to spool it to disk from the buffers, the proxy just aborts the caching operation. This will tend to favor storing data from far away or slow servers in the cache, while data from fast, nearby servers is fetched on each request.[1]
- The proxy can record all the gaps and continue reading from the tap. Later, whenever it finds spare cycles, it can fill the gaps by issuing additional requests for the document to the server using the Range Request header with a conditional GET [4] to retrieve only the missing parts.

We have not tested TCP Tap against a real HTTP cache workload, but insights derived from the TCP Splice performance measurements suggest that the probability of a tap buffer overflow will be low. Figure 3 shows that in identical tests the splice proxy is only 55% utilized when the application layer proxy is CPU constrained. A TCP Tap proxy carrying the same load as a current application layer proxy should always have free CPU cycles to drain the tap buffer. Choosing a large tap buffer, (e.g., 2*socket buffer size), will further minimize the chances of buffer overflow. It will always be possible to avoid overflows by throttling system throughput, but we suspect that turning off caching instead will be an overall better policy.

5 Discussion

Combining TCP Splice and TCP Tap opens the possibility of building a high performance HTTP caching proxy controlled by an application layer proxy. Though we haven't finished our current implementation yet, we are optimistic that an HTTP proxy built using these techniques will outperform proxies built via conventional application layer techniques. Our results from the application layer firewall performance measurements already show a potential for improvement; TCP Splice saves processing cycles as well as buffering and clearly demonstrates the opportunity to support higher throughput and more connections.

We have designed TCP Splice and TCP Tap to have three main benefits when deployed in proxy systems. The first is a decrease in system overhead and resource requirements. The second is a reduction in the latency the proxy system imposes on end-to-end transfer times, and the third is a simple API to ease integration into proxy systems.

TCP Splice reduces system overhead in several ways. Clark [3] reports that the cost of operating a TCP connection is dominated by the per-byte operations and the operating system support, such as timers and memory buffers (mbufs). TCP Splice eliminates two CPU passes over every byte of data transfered by eliminating the application layer relay that requires copying the data from kernel to user space and back again. It eliminates another two CPU passes over all the transferred data bytes by replacing the TCP checksum operations that have to be performed when the packet is received and forwarded by an application layer proxy with a checksum update operation. Advanced buffer management techniques [2] could be used to eliminate the data copies, but unless TCP Splice is in use the proxy must still incur the overhead of checksumming the data twice and of running two TCP state machines with their associated buffers, sockets, and timers.

TCP Tap reduces resource requirements by allowing the system to close or free resources once the splice and tap are set up. While a proxy using TCP Tap still requires two sockets and four socket buffers (two for each socket) to setup a splice to relay data, after the splice is completed the requirement drops to one buffer and one socket to implement and read from the TCP Tap buffer. A conventional proxy requires the two sockets and four socket buffers for the life of the logical connection. Furthermore, once the TCP Splice completes, the timers and protocol state machine associated with the proxy-server and proxy-client connections are no longer needed and are freed.

Minimizing the latency of packets traveling between the client and server is important as increased latency acts to decrease throughput and is potentially visible to the end-user as a decreased response time. All proxies increase the latency of packets traveling between the client and server, since the proxy must relay the packet at some layer of the network stack. Running at the top of the network stack, application layer proxies add the greatest amount of latency and have the largest variability in latency. The per-packet processing overhead of splicing a packet from one connection to another is constant and small, so proxies using TCP Splice add less overall latency to the logical connection than other techniques. Additionally, proxy applications using TCP Tap can actually fall behind in reading the data being transfered through them without adding latency to the end-to-end connection, because the data packets are still spliced and forwarded as soon as they arrive. Conventional application layer proxies typically stall the data flowing end-to-end whenever they can not read data as quickly as it arrives.

TCP Tap was designed to be compatible with the existing Berkeley Sockets API in order to ease integration with both present and future proxies. While there is a trend towards router vendors implementing proxy functionality on top of their proprietary operating systems and hardware for performance reasons, most application layer proxies today are built on top of general purpose operating systems for the low cost, flexibility and opportunity for innovation that such designs allow. Perhaps this is an indication that minimizing changes to API is crucial for the success of any optimization method. We have made minimal changes to the socket API to maximize the ability of programmers to include TCP Splice and TCP Tap in their proxies.

Recent studies of web server performance [7, 10] have shown that WWW servers spend most of their time in the kernel. A significant part of this time is spent in system call processing and copying data across the user-kernel protection boundary. A web sever makes at least fourteen system calls to process even the smallest GET request [7], thus reducing system call overhead is a commonly used technique for improv-

[1] There is the potential for a group failure if the proxy starts many spool-to-disk operations, and then falls behind on all of them so that none can be completed and all the partial data on disk must be thrown out. We do no yet have the experimental data to evaluate this strategy.

ing web server and proxy performance. Microsoft, for example, has introduced two new socket layer calls, `acceptex()` and `transmitfile()` in Windows NT. `acceptex()` combines the separate `accept()`, `getsocketname()`, and `recv()` calls normally made to begin processing a request into a single system call, while `transmitfile()` directs the kernel to move the contents of a file straight into a network connection, bypassing the `read()` and `write()` tight loop. Our proposal is complementary to the methods developed by the high performance API community, and extends the savings to more parts of a proxy's function. The benefits of the `acceptex()` system call can be easily combined with the TCP Splice and TCP Tap approach. TCP Splice provides performance advantage by making forwarding a zero-copy operation and TCP Tap provides a light-weight one-copy method of caching the forwarded stream.

Though we only report performance results for large size transfers, we expect the relative benefits of TCP Splice over application layer forwarding to also hold for a mixed workload consisting of small, medium, and large transfers. We are in the process of conducting more realistic experiments to verify our hypothesis. If experimentation shows a minimum size transfer for which TCP Splice and TCP Tap give improved performance, it would be trivial to have the proxy read the response headers for the the Content-Length field and only splice larger connections. This once again illustrates the basic strength of TCP Splice and TCP Tap, which is the manor they work together to provide flexible primitives for improving the performance of a wide range of application layer proxies.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. Internet Request For Comments RFC 1945, May 1996.

[2] Jose Carlos Brustoloni. User-level protocol servers with kernel-level performance. In *IEEE INFOCOM'98: Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 463–471, San Francisco, CA, April 1998. IEEE. Available as http://www.cs.cmu.edu/afs/cs/user-/jcb/papers/infocom98.ps.

[3] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Internet Request For Comments RFC 2068, January 1997.

[5] David A. Maltz and Pravin Bhagwat. MSOCKS: an architecture for transport layer mobility. In *IEEE INFOCOM'98: Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1037–1045, San Francisco, CA, April 1998. IEEE. Available from http://www.cs.cmu.edu/∼dmaltz.

[6] David A. Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. Technical Report RC 21139, IBM, March 1998. Available from http://www.cs.cmu.edu/∼dmaltz.

[7] Erich Nahum, Tsipora Barzilai, and Dilip Kandlur. Evaluating high performance socket api's on aix. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, February 1998. Submitted for publication.

[8] Paul Resnick and Jim Miller. PICS: internet access controls without censorship. *Communications of the ACM*, 39(10):87–93, 1996.

[9] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, The Implementation*, volume 2. Addison-Welsley, 1995.

[10] David Yates, Virgilio Almeida, and Jussara M Almeida. On the Interaction Between an Operating System and Web Server. Technical Report CS 97-012, Boston University, Computer Science Dept., Boston, MA, July 1997.

[11] Bruce Zenel and Dan Duchamp. General purpose proxies: Solved and unsolved problems. In *Proceedings of Hot-OS VI*, May 1997. Read as http://www.mcl.cs.columbia.edu/∼baz/ps/hot-os-vi.ps.