Optimizing TCP Forwarder Performance

Oliver Spatscheck, Jørgen S. Hansen
John H. Hartman and Larry L. Peterson

TR 98-01

Abstract

A TCP forwarder is a network node that establishes and forwards data between a pair of TCP connections. For example, a firewall that places a proxy between a TCP connection to an external host and a TCP connection to an internal host—for the purpose of implementing access control to a resource on the internal host—is an example of a TCP forwarder. Once the proxy approves the access, it simply forwards data from one connection to the other. We use the term *TCP forwarding* to describe indirect TCP communication via a proxy in general. This paper characterizes the behavior of TCP forwarding, and illustrates the role TCP forwarding plays in common network services like firewalls and HTTP proxies. We introduce an optimization technique, called *connection splicing*, that can applied to a TCP forwarder, and reports the results of a performance study designed to evaluate its impact. Connection splicing has the effect of improving the performance of TCP forwarding by a factor of two to four, making it competitive with the performance of an IP router running on the same hardware.

February 9, 1998

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

It is increasingly common that processes communicate with each other indirectly through a proxy. This happens, for example, in a firewall where a proxy mediates the flow of information between a TCP connection to an untrusted external entity and a TCP connection to a trusted local entity. We use the term *TCP forwarding* to denote the general pattern of indirect communication over a pair of TCP connections via a proxy.

One consequence of TCP forwarding is that there is often a single network node—e.g., a firewall—that runs proxies on behalf of many different indirect communications. This network node, which we call a *TCP forwarder*, plays a role very similar to that of an IP router, except it must execute two TCP endpoints and a proxy for every "flow" that passes through it. The performance of this TCP forwarder—i.e., its throughput in terms of packets-per-second—can play a significant role in the network performance perceived by the communicating entities.

This paper makes three contributions. First, it defines a general framework for TCP forwarding, and demonstrates the relevance of TCP forwarding to three applications: firewalls, HTTP proxies, and process migration. Second, it describes an optimization technique, called *connection splicing*, that can be used to improve the performance of a TCP forwarder. An implementation of connection splicing in the Scout operating system is also presented. Third, it reports the results of a performance study that measures the effectiveness of connection splicing. The study shows that connection splicing can improve performance of a TCP forwarder by a factor of two to four, bringing its performance close to that of an IP router implemented on the same platform.

2 TCP Forwarding

It is often useful for two entities to communicate indirectly through two separate TCP connections, one for each entity, rather than directly through a single TCP connection. An entity called a *proxy* mediates the communication, interposed between the two connections, and controls the flow of data between the communicating parties (Figure 1). The proxy runs the show—it decides if the parties can communicate, and if so, what is communicated. A proxy can both restrict and enhance the communication. For example, a telnet proxy can restrict to which computers the outside world may connect, and perhaps which users may log in. On the other hand, a telnet proxy could also serve as a clearinghouse for a collection of servers by providing a single connection point for outside telnet accesses. The telnet requests are processed by the proxy and forwarded to the appropriate computer, shielding the outside world from the internal structure of the site.

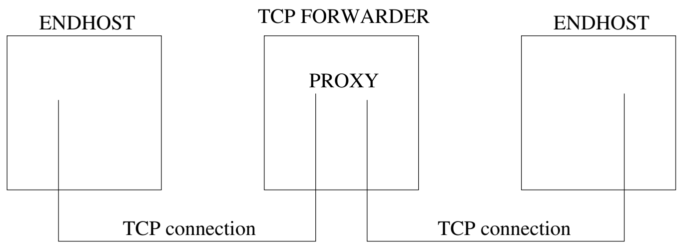

Figure 1: TCP forwarding via a proxy.

We use the term *TCP forwarding* to refer to communication relayed over two TCP connections via a proxy. TCP forwarding is not as simple as copying bytes from one connection to the other, however. The proxy must control the communication as well as relay bytes, and therefore, a proxy has two modes: *control mode* and *forwarding mode*. In control mode the proxy processes either out-of-band or in-band control information. Once the control functions have been completed, the proxy switches to forwarding mode to move data between the connections. After the data transfer, the proxy may switch back to control mode. For example, a telnet proxy starts off in control mode, and processes a telnet request to determine if the connection should be allowed, based on the target machine, port, and perhaps user ID. Once the connection has been completed, the proxy switches into forwarding mode to transfer data between the two computers. Switching between these two modes of operation is the primary difficulty in developing an optimized TCP forwarding mechanism.

The processing that is done in control mode varies greatly between proxies, from very little during connection setup to continuous monitoring of the data stream while forwarding to extract control information. Proxies can be broadly classified into four categories, depending on the degree of control processing they do. Proxies that perform a minimum of control processing are those that only restrict and route based on IP addresses and port numbers. They are in control mode only during connection setup; after that they switch to forwarding mode for the duration of the connection. An FTP proxy is an example: it processes an FTP request in control mode on the control connection, sets up a data connection between the two computers, and switches to forwarding mode on that data connection until it is closed.

The second class of proxies performs more control processing because they authenticate the user or request and base routing decisions on either the result of the authentication or control information passed in the TCP connection. A telnet proxy is a member of this class. Typically, a telnet proxy requests a user ID, password, and the destination of the telnet request. This information is received on the TCP connection by the proxy and is used to authenticate the user and establish a connection to the correct remote machine. At this point the proxy simply forwards data between the two connections.

The third class of proxies remains in control mode for all data transferred in one direction, but switch to forwarding mode for data transferred in the other. An example is an HTTP proxy that processes the HTTP requests (control information) sent by clients, but simply forwards the data returned by the HTTP server.

The fourth class remains in control mode and continuously monitors data passed in both directions. This might be the case for a proxy that allows users on a protected network to access HTTP servers on the Internet. The proxy could filter outgoing accesses to restrict the servers that can be reached, and filter incoming access responses to remove (untrusted) Java code.

TCP forwarding has many uses, including such diverse functions as a network firewall, an HTTP proxy, and a process migration system. These three examples illustrate the power of TCP forwarding, and the necessity for an efficient mechanism for doing so.

2.1 Firewall

A firewall provides limited connectivity between a protected network and the relative chaos of the Internet, as shown in Figure 2. The firewall contains different types of proxies, each handling a different type of communication between the two networks, such as telnet, FTP, etc. A typical proxy accepts connections on one network, authenticates the entity making the connection request, and forwards the data to the other network, perhaps after applying a filter. The firewall either uses its own IP address (*classical proxy*) or is completely transparent to the user (*transparent proxy*) [5]. A classical proxy must use the control information in the request to determine the connection's true destination.

2.2 HTTP Server Proxy

TCP forwarding can also be used to develop scalable servers such as HTTP servers. HTTP server names are embedded in the URL namespace, making it difficult to implement a single HTTP service from a collection of servers. Load-balancing across the collection is a problem; Web sites typically offer the users a selection of servers from which to choose, manipulate the DNS mappings to change dynamically the IP address associated with a site name, or use the HTTP redirection mechanism to redirect requests to unloaded servers. The first two offer coarse-grained load balancing, while the last requires two HTTP connections per URL accessed.

An HTTP server proxy that forwards TCP connections is a better solution. Clients connect to the proxy, which processes their requests and forwards them to the appropriate server. The proxy must continually monitor the data received from the clients, however, so that requests can be extracted and processed, and the connections re-forwarded as appropriate. The data returned from the servers, however, is simply forwarded to the clients.

Such an HTTP proxy might implement a variety of forwarding policies, in addition to load-balancing over a set of homogeneous servers. The proxy could forward connections to servers based on the URL requested, allowing a

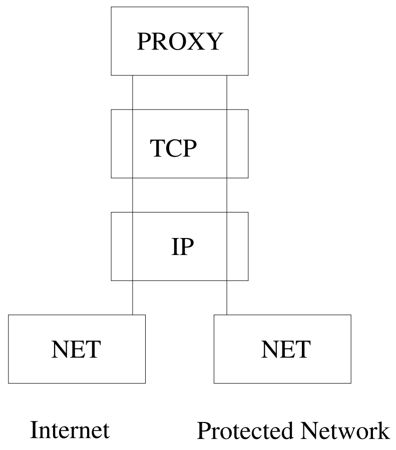

Figure 2: Overview of a application-level firewall. Data from one network pass through the proxy which forwards them to the other network if the desired security guarantees are not violated.

collection of servers, each of which serves a different collection of pages, to appear as a single site. It could also provide more complex functionalities as described in [4].

2.3 Process Migration

The final example, perhaps a bit more esoteric, is that of transparent process migration. Process migration is the technique of moving a running process from one computer to another. Different levels of transparency are possible, of course, but ideally process migration should be completely transparent, so that aside from perhaps a performance difference, a migrated process cannot detect that it has changed computers. The techniques used to implement process migration are beyond the scope of this paper, but one difficulty is how a migrated process interacts with the outside world. After a process migrates, which IP address should be used to contact it? If the IP address of the new host computer must be specified, then migration isn't transparent to the outside world. If the IP address of the original host computer is used then obviously packets arrive at the wrong machine. TCP forwarding allows the original host to forward packets to the new host, so that both the migrated process and the outside world are unaware of the migration. All that is needed is a simple proxy on the original host that merely relays data in forwarding mode once a connection has been established.

3 Connection Splicing

This section describes an optimization technique, called *connection splicing*, that improves TCP forwarding performance. It includes a discussion of the many complications that make connection splicing difficult in practice. To simplify the following discussion, we focus on the flow of data in a single direction; the same work must also be done for data going in the other direction.

3.1 Overview

The proxy involved in TCP forwarding operates in either control mode or forwarding mode. The basic idea of connection splicing is to detect when a proxy makes a transition from control mode to forwarding mode, and then splice the two TCP connections together into a single forwarding path through the system. The resulting spliced connection replaces the processing steps (and associated state) required by two TCP connections with a single reduced processing step (and associated state).

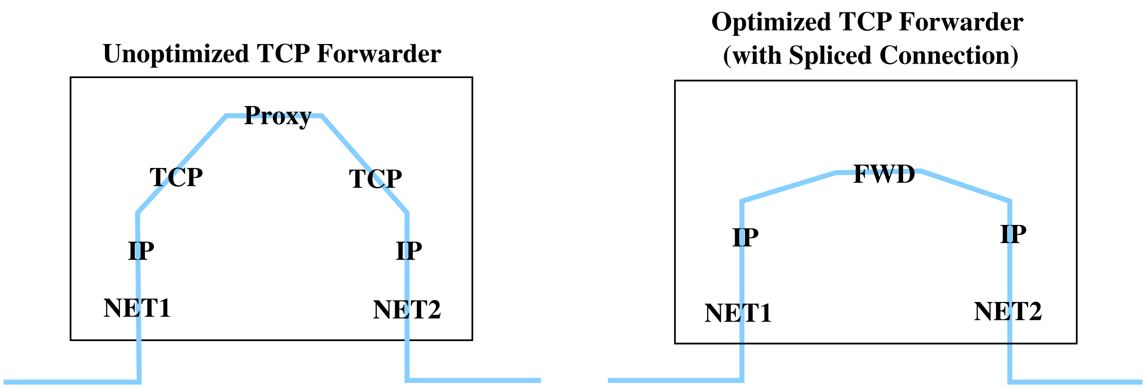

Figure 3: Optimizing two TCP connections into a single spliced connection.

Figure 3 schematically depicts the optimization. The standard (unoptimized) forwarder on the left requires TCP segments to traverse TCP twice, with each instance of TCP maintaining the full state of the connection. In this case, the Proxy simply passes segments from one connection to the other when it is in forwarding mode. The optimized forwarder on the right replaces the Proxy and two TCP processing steps with a single FWD processing step. FWD maintains just enough state to successfully forward TCP segments from one network to another. The state FWD needs to maintain is described later in this section.

A single proxy might require both configurations, however. The configuration on the left *must* exist when the proxy is in control mode; the proxy must be in the loop because it needs to inspect the data flowing between the two TCP connections. The configuration on right *may* exist while the proxy is in forwarding mode. Forwarding can also happen in the left configuration, but performance suffers. With this perspective in mind, there are three cases to consider: how optimized configuration on the right works in the steady state (Section 3.2), how the system makes the transition from the left-hand configuration to the right-hand configuration (Section 3.3), and how the system makes the transition from the right-hand configuration back to the left-hand configuration (Section 3.4).

Typically TCP forwarding starts in the unoptimized configuration, makes a transition to the optimized configuration when the proxy shifts from control to forwarding mode, and sometimes reverts back to the unoptimized configuration should TCP forwarding go back to control mode. Note that while the connection splicing optimization is in effect, the two independent TCP connections shown on the left no longer exist on the forwarder.

3.2 Forwarding

The primary task of the FWD processing step shown in Figure 3 is to change the header of incoming TCP segments to account for the differences in the two original TCP connections. Since the two TCP connections were established independently, their respective port numbers and sequence numbers are probably different. The IP addresses associated with the connections might also differ, resulting in changes that affect the IP pseudo header as well.

Figure 4 depicts the TCP segment header; the boldface fields are those that FWD modifies. The following outlines the transformations FWD applies to each segment it forwards from one connection (A) to another connection (B). For now, we ignore the problem of moving a TCP forwarder into the optimized state, and focus instead on the work involved in forwarding segments once FWD is in place. Also, we assume that the two TCP connections were established independently. If their establishment was in fact interleaved—so that one connection knew what port and sequence numbers were being used by the other connection—then additional optimizations are possible, as described in section 3.6.

Port numbers: If the TCP forwarder operates as a classical proxy, the port numbers of both TCP connections will probably differ. Therefore, the source and destination port numbers of segments arriving on A have to be

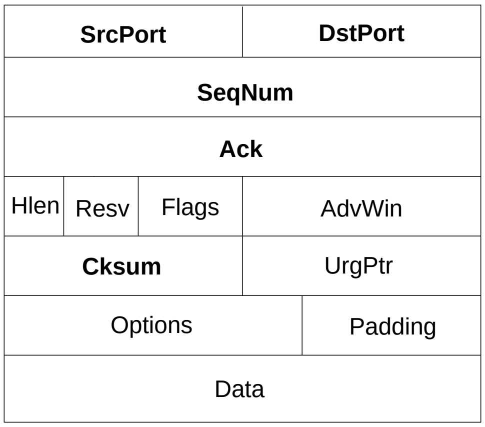

Figure 4: TCP segment header with fields modified by FWD in bold.

changed to the port numbers of connection B. If the TCP forwarder is a transparent proxy, this change is unnecessary because the proxy uses the same port numbers as the server.

Sequence Number: The sequence number used by segments received by FWD on A are probably different from those used for segments sent by FWD on B. This is because TCP initializes sequence numbers randomly for each independent connection. The sequence number for an outgoing segment is computed by adding a fixed offset to the sequence number in the incoming segment.

Acknowledgment Number: The acknowledgment number acknowledges the sequence numbers forwarded *in the other direction*. Thus the acknowledgment number in an outgoing segment is computed by subtracting from the sequence number in the incoming segment the sequence number offset for segments flowing in the other direction.

Checksum: Modifying the other fields requires adjusting the TCP checksum. A constant checksum patch representing the "delta" in the checksum is used to do this efficiently. If the FWD acts as a classical proxy, the changes to the IP address fields in the IP pseudo-header are also reflected in this checksum patch.

The following pseudo code describes the changes to a segment transferred from A to B. All header fields marked Input represent the segment header values in the received segment. The header fields marked Output represent the segment header values used in the outgoing segment. Bold variables indicate constants that are part of FWD's state. Subscripts indicate the direction for which these constants are used; e.g., **SeqNumOffset**$_{A \rightarrow B}$ represents the sequence number offset used to patch sequence numbers on segments received from A and sent to B.

Output.DstPort = **RemotePort**$_B$
Output.SrcPort = **LocalPort**$_B$
Output.SeqNum = Input.SeqNum + **SeqNumOffset**$_{A \rightarrow B}$
Output.Ack = Input.Ack - **SeqNumOffset**$_{B \rightarrow A}$
Output.Cksum = Input.Cksum + **CksumPatch**$_{A \rightarrow B}$

The checksum calculation shown in the pseudo code is more complicated than simple addition. To account for overflows or underflows during sequence number and acknowledgment number calculations it is necessary to add or subtract one from the checksum. This is because the checksum is the one's complement of the one's complement sum of the segment.

Splicing two TCP connections significantly changes the behavior of the forwarding proxy. In the unspliced case segments sent to the proxy are acknowledged when they are processed by the incoming TCP stack. The proxy then takes responsibility for the data, resending them as necessary to ensure they reach their destination. Data are buffered in the outgoing TCP stack until they are acknowledged by the destination. When the two connections are spliced the segments no longer traverse the two TCP protocol stacks. The proxy doesn't acknowledge data coming from the sender, nor does it resend data to the destination. Data and acknowledgements are forwarded without processing, requiring the two endpoints to handle retransmission and reordering.

Forwarding segments requires internal state in FWD. Some of this state is required to modify the header fields, such as the port numbers, sequence offsets, and checksum patch. FWD must also detect reset or termination of the TCP connection. To do so it parses the flags in the header and keeps a simplified TCP state machine. FWD also keeps one timer that is used to timeout the connections; all other TCP timers are not used.

If it is possible that the optimized forwarder will revert back to a standard forwarder, FWD also needs to store the current advertised window, the highest sequence number sent, and the highest ACK seen that will fit in the advertised window. The process of converting a spliced connection back to an unoptimized TCP forwarder is discussed in Section 3.4.

3.3 Splicing

The header modifications required to forward a segment are relatively straight-forward; the more difficult task is transitioning from the unspliced state to the spliced state. The difficulty is caused by acknowledged data buffered in the forwarder. This data might be buffered by the receiving TCP's receive buffer, within the proxy itself, and in the sending TCP's send buffer (Figure 5). The acknowledged data must be reliably forwarded to its destination. This data also influences the offsets calculations required by the spliced connection.

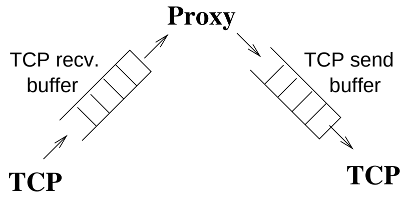

Figure 5: TCP buffers potentially containing acknowledged data.

First, all data acknowledged by either connection on the unoptimized TCP forwarder must be reliably delivered to its destination. The important point is that this data has already been acknowledged by the forwarder, so it cannot depend on the source host to take responsibility for possibly retransmitting the data in the future. Thus, the forwarder must continue to run TCP—the only way it can reliably deliver data—until the currently buffered (acknowledged) data are reliably delivered to the destination. During the time this data are being drained, however, new segments may arrive. The forwarder obviously cannot let TCP acknowledge this data, because doing so will just give it even more data to reliably deliver, and it is impractical to wait until the two connections go idle before completing the splice. Fortunately, there are two ways to handle newly arriving segments during this transition period.

The first option is to delay the activation of the spliced connection until after the buffers have drained. During this time, a limited amount of new segments that arrives is delivered to TCP (for acknowledgment processing only) and held in a separate buffer for FWD; they are not acknowledged and they are not placed in the incoming connection's receiver buffer. If the buffer overflows while TCP is still processing acknowledgments, the segments are dropped after the acknowledgements are processed. When the transition is complete, this buffered data are processed by FWD as though they had just arrived. Again the TCP protocols are suspended as soon as all buffers are drained. This solution may drop data if the FWD buffers overflow while the TCP buffers are being drained. If the amount of data buffered in TCP is small, then the FWD buffers are unlikely to overflow.

The second option is to allow FWD to begin forwarding data concurrently with draining the buffers. Should any new data arrive during the transition, it is important that the original TCP protocols not acknowledge this new data; they are only allowed to process the acknowledgments contained in those segments so that the buffers drain. In other words, all newly arriving segments are delivered to both the original TCP protocol (for acknowledgment processing only) and to FWD (for forwarding to the receiver). This solution does not drop data, but may cause data to be delivered out-of-order. Segments processed by FWD may be delivered before segments traversing the original TCP connections. This will not affect correctness because the destination will reorder the segments.

During the time that FWD operates concurrently with the draining process, both forwarded segments and drained segments will arrive at the destination. This means it is possible that the TCP draining buffers on the forwarder might receive an acknowledgement for a sequence number that is larger than maximum sequence number in its send buffer. This acknowledgement is really meant for the source host, but since the forwarder is still processing acknowledgements in an attempt to drain its buffers, it will receive this acknowledgement too. To allow for this possibility, TCP running on the forwarder during the transition must be able to accept acknowledgments up to one full window size larger then the maximum sequence number in its send buffer.

The second thing that must be done during splicing is to initialize the internal state of FWD. This requires computing the sequence number offsets (**SeqNumOffset**$_{A\rightarrow B}$ and **SeqNumOffset**$_{B\rightarrow A}$) and the checksum patches (**CksumPatch**$_{A\rightarrow B}$ and **CksumPatch**$_{B\rightarrow A}$) used by FWD. The sequence number offsets can be calculated as soon as all acknowledged data have been drained. If acknowledged data still exist in one of the forwarder's buffers, then it is necessary to subtract the length of this buffered data from the corresponding sequence number offset. This is because the sender of a segment that is directly forwarded assumes that the buffered data was deliverd, and therefore, the sequence number of the source's TCP protocol has already been increased. The checksum patch can be calculated as soon as the other offsets are known since the changes in port number and IP address are already known.

3.4 Unsplicing

When the forwarding proxy switches from forwarding mode to control mode the connections must be unspliced. There are two complications; the first is to be able to detect that it is necessary to switch back to the unoptimized state; i.e., that the forwarder has moved from forwarding mode to control mode. The second is to correctly make the transition. The solutions to these two complications are intertwined.

It may be difficult to decide when the proxy should switch back to control mode. If the control information is sent over the spliced connection the proxy has to monitor the data being forwarded to detect the control information. This is difficult because the FWD protocol does not reorder the segments it receives, nor does it buffer segments. The proxy has to find the control information by looking at out-of-order segments, one at a time. This makes it unlikely that the proxy will be able to filter the data to find control information. However, it seems useful to trigger a switch back in unoptimized mode as soon as data are transmitted in a certain direction. An HTTP 1.1 proxy, for example, might allow the forwarding of HTTP replies but want to examine all, possibly pipelined, HTTP 1.1 requests.

Dealing with acknowledgements makes it difficult to unsplice a connection. When the forwarder reverts to two TCP connections and proxy it must take over handling acknowledgements. If there are no unacknowledged segments outstanding on the spliced connection, the transition back to unspliced is easy. The re-constructed TCP connections are initialized with the sequence numbers, acknowledgment numbers, and advertised window sizes stored as FWD state. The state-machine is progressed to the current state, the timers and the send window are initialized with their initial values, and a slow start is initiated. This makes it possible to stop instantly forwarding new segments, but will require the retransmission of lost segments.

If there are outstanding unacknowledged segments, however, the forwarder must either wait for them all to be acknowledged—dropping data if necessary and then switch as described above—or else continuously monitor the segment stream until it has copies of all unacknowledged segments. It then uses this information to initialize the TCP connections and buffers. This solution does not drop any segments, but up to two full window sizes might have to be buffered before the switch over can be completed.

3.5 Flow Control

During unoptimized operation flow control is handled by the two independent TCP protocols on the forwarder, and the TCP protocol on the end hosts. During optimized operation, flow control is handled by the end hosts only; the forwarder merely drops segments, just as a congested router drops IP datagrams.

There is a complication during the transition to a spliced connection, however. Shortly after the switch to the spliced connection, the advertised window might be either too big or too small. For example, the window advertised by the source host to the forwarder might be smaller than the window advertised by the forwarder to the destination host. In this case, the destination host will suddenly see a smaller advertised window after the connection is spliced, possibly triggering unnecessary retransmissions. Similarly, the send window of a host might also be bigger than the advertised window of its new peer. In this case, it is very likely that data will be transmitted unnecessarily.

More subtly, the send window of both end hosts might not represent the bandwidth of the link. If the send window is too big, the host will send too many segments and generate unnecessary congestion. However, this can only happen if traffic is extremely bursty. Otherwise, the limited buffer space available on the TCP forwarder should synchronize the send window sizes of both TCP connections.

3.6 Additional Optimizations

The connection splicing optimization can be applied not only at the TCP level, but also to unfragmented IP datagrams. In addition, the optimization can be applied to the first IP fragment of an IP datagram if we allow the unfiltered forwarding of all remaining fragments, and if the MTU is large enough so that the first fragment will contain the TCP segment header.

In these two cases, the forwarder can process the IP datagrams similarly to an IP router, with the additional TCP segment header manipulation described in the previous section. Figure 6 illustrates this scenario, which we denote as a combined IP/FWD processing step. The important consequence of being able to forward TCP segments at the IP level is that it makes it possible to apply any of the optimizations one might apply to an IP router. For example, if the forwarder is connected to two Ethernets, we can modify and forward the Ethernet packets directly.

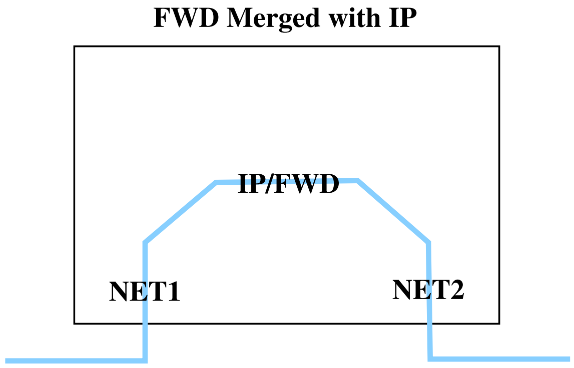

Figure 6: Further optimizing the spliced connection when there is no fragmentation.

Finally, under certain circumstances it possible that the TCP forwarder can tolerate the unfiltered forwarding of all IP fragments, that is, FWD implements the identity transformation. This would happen if the unoptimized forwarder is configured as a gateway intercepting TCP connections and was careful in selecting port numbers and the starting sequence numbers when the original pair of TCP connections were opened. This being the case, FWD can be omitted and the TCP forwarder operates just like an IP router.

4 Connection Splicing in Scout

Connection splicing can be implemented in any operating system, but the technique is particularly effective in a communication-oriented operating system such as Scout[8]. Scout is a configurable OS explicitly designed to support data flows, such as video streams through an MPEG player, or a pair of TCP connections through a firewall. Specifically, Scout defines a *path* abstraction that encapsulates data as it moves through the system, for example, from input device to output device. In effect, a Scout path is an extension of a network connection through the OS. Each path is an object that encapsulates two important elements: (1) it defines the sequence of code modules that are applied to the data as it moves through the system, and (2) it represents the entity that is scheduled for execution.

The path abstraction lends itself to a natural implementation of TCP forwarding. Figure 7 schematically depicts a naive implementation of TCP forwarding (the unoptimized case) in Scout. It consists of two paths: one connecting the first network interface to the proxy and another connecting the proxy to a second network interface. In this figure, the path has a source and a sink queue, and is labeled with the sequence of software modules that define how the path "transforms" the data it carries.[1] To a first approximation, the configuration of Scout shown in Figure 7 represents the implementation one would expect in a traditional OS.

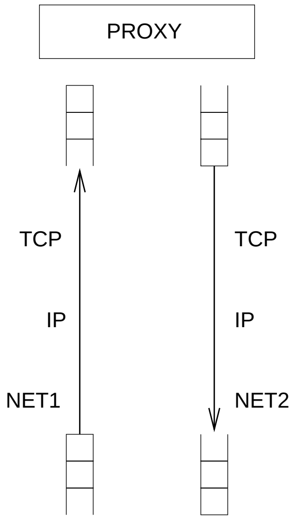

Figure 7: TCP forwarding implemented in two Scout paths.

The two-path configuration shown in Figure 7 has suboptimal performance because it requires a handoff of each incoming segment from the first path to the proxy, and then from the proxy to the second path. In Scout, the entire device-to-device data flow can be encapsulated in a single path (Figure 8). This is the implementation of choice for the unoptimized TCP forwarding case in Scout.

Connection splicing is then implemented within the same framework. Figure 9 illustrates the two optimized configurations discussed in Section 3: the path on the left corresponds to the right-hand case from Figure 3, while the path on the right corresponds to the case shown in Figure 6. Note that the right-hand path looks very much like an IP router would in Scout.

Looking at the implementation in a bit more detail, each path consists of a linked list of *stages*, where each module that the path traverses contributes a stage to the path during path creation. Abstractly, each stage contains the path-specific code and state for the corresponding module; e.g., the TCP control block is contained in the TCP stage of the paths shown in Figures 7 and 8. When the proxy in an unoptimized TCP forwarding path detects a transition to forwarding mode, it does five things:

- Stops processing incoming segments and allows segments to accumulate in the path's input queue.
- Unlinks the two TCP stages and the proxy stage from the path and replaces them with a preliminary FWD stage.
- Continues processing incoming segments and data in the TCP buffers until the TCP buffers are drained.
- Unlinks the preliminary FWD stage and replaces it with the final FWD stage.

[1] As in Section 3, we focus on data flowing in one direction. In reality, Scout paths, like TCP, support bi-directional data flows.

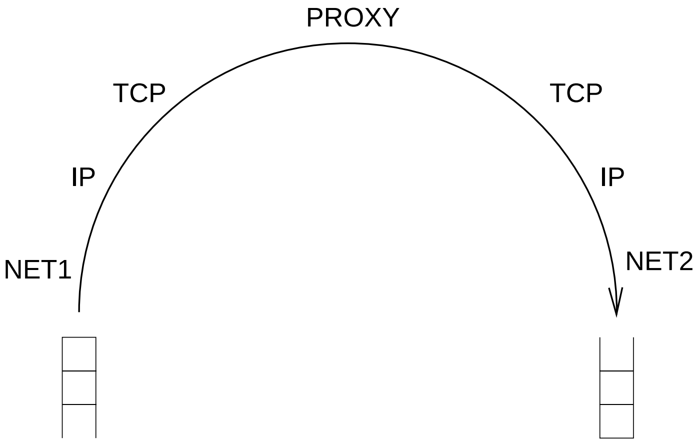

Figure 8: TCP forwarding implemented in a single Scout path.

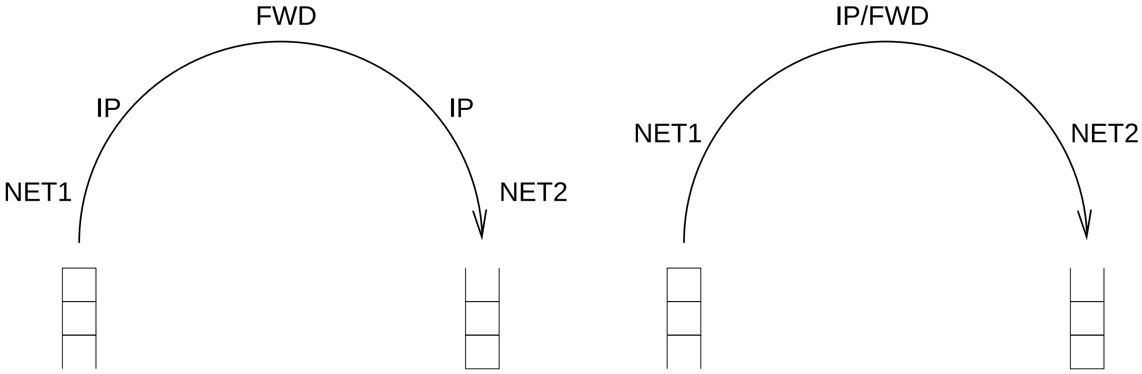

Figure 9: Connection spliced paths in Scout.

- Continues processing incoming segments.

The difference between the preliminary FWD stage and the final FWD stage is that the former forwards the segments and reliably drains the TCP buffers, whereas the latter only adjusts segment header fields.

One subtlety is that there are seldom any segments queued *within* the path that need to be drained; Scout is non-preemptable, so in practice once a segment is removed from the input queue it is processed completely and deposited in the output queue. The only time a segment gets buffered in the middle of a path is when the scheduler selects the path for execution, the segment makes it as far as the outgoing TCP stage, but the advertised window on the second connection is closed. It would be possible to take the outgoing window into account when making the scheduling decision—i.e., not schedule a TCP forwarding path until it was certain that the segment could make it all the way to the output queue—but the consequence is that the segment would remain in the input queue, and thus, not acknowledged on the incoming TCP connection.

There is one final issue to consider: how Scout classifies each incoming packet to determine the path to which it belongs. Scout classifies packets by inspecting various header fields, such as ETH's type field, IP's protnum field, and TCP's port fields. While the details are beyond the scope of this paper, the relevance to connection splicing is that even after an unoptimized TCP forwarding path is spliced, the classification machinery remains the same. In other words, the spliced path no longer does any TCP processing, but the TCP port fields are still used to classify packets for the spliced path.

5 Performance

This section provides measurements of the effect of connection splicing on TCP forwarding. To make the study concrete—and to give us an existing system against which we can compare our approach—we focus on a simple firewall configuration. The proxy in the firewall does not perform any processing in control mode; it is always in forwarding mode.

5.1 Test Cases

We measured the following configurations of Scout:

2-Path: This is a full blown TCP forwarder, as depicted in Figure 7. This TCP forwarder uses two separate TCP paths meeting at the proxy—one to each network device. As going from one path to another often will require a context switch, this configuration is the closest to the structure of a firewall in a regular operating system like Unix or NT.

1-Path: This is the configuration shown in Figure 8. This case is similar to the 2-path configuration, except the two network devices are connected by a single path. This is the natural way of expressing a TCP forwarder in Scout. Note that this configuration still involves two separate TCP connections but a single Scout path.

FWD: This is an optimized version of 1-Path. Here the TCP connections have been spliced into a single connection, and the forwarder is reduced to updating the TCP headers. This configuration still supports reassembly of IP packets. This case corresponds to the left-hand configuration in Figure 9.

IP/FWD: This is a further optimized version of **FWD**. The network level packets are modified directly and forwarded. As a consequence, this configuration does not support reassembly of IP packets. This is the case corresponds to the right-hand configuration in Figure 9.

IP Router: This is an IP router. It also modifies network packets directly in the same way as **IP/FWD**, but it does not update TCP headers. It is included to show the lowest possible overhead for an intermediate host in Scout.

To compare the Scout performance with a more general-purpose operating system, we also measured the performance of a firewall and IP routing on Linux. We compiled the Linux kernel to optimize for IP routing. We consider three configurations:

TIS Firewall: The TIS firewall toolkit offers full filter functionality.[9]. We have configured it to use a null filter (plug-gw).

Filtering IP Router: The in-kernel Linux IP forwarding has support for filtering on IP addresses, protocol numbers and port numbers. This is the closest thing in Linux to the IP/FWD case in Scout, except Linux neither permits starting with a proxy and later dynamically switching to the spliced connection, nor updating TCP headers.

IP Router: This is the basic in-kernel Linux IP forwarding with no filtering. This shows the lowest possible overhead of the Linux configuration.

Finally, we measure the performance of two machines connected back-to-back to evaluate the overhead of injecting a third host on the network path.

All hosts used in our experiment are 200 MHz PentiumPro workstations with 256KB cache, 128MB ram, and Digital Fast EtherWORKS PCI 10/100 (DE500) 32-bit PCI 10/100 Mb/s adapters. The Linux version used was 2.0.30. The physical configuration of our test setup is shown in Figure 10. To saturate the network during throughput tests, we connected three hosts on each side of the firewall. All test are performed between a server (hosts S1 to S3) and a client (hosts C1 to C3). In the back-to-back case, the setup was modified by connecting the two hubs to each other. All servers and clients were running Scout, as the lower complexity of Scout resulted in less variation in the measurements than Linux.

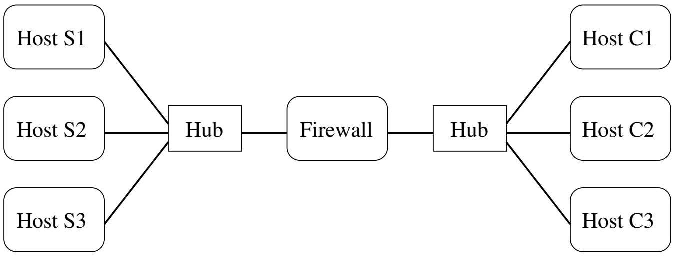

Figure 10: Test Setup

5.2 Results

For all configurations, we measure the per-packet processing time for small (1-byte) and large (1460-byte) segments, and the aggregate throughput achieved with multiple connections. For Scout, we also measure the time it takes to switch from unoptimized to optimized.

5.2.1 Processing Overhead

To measure the per-packet processing overhead, we measured the packet round-trip times for 10,000 packets, and subtracted the back-to-back latency and network interface latency. The subtracted components are summarized in Table 1. The network interface latency was obtained by measuring the processing time of a packet in the IP router configuration—that is, the time from when the packet is removed from the network interface by the interrupt handler to the time it inserted into the transmit queue of the other network interface—and subtracting this time from the total latency added by the router.

Latency		1-byte TCP Segment	1460-byte TCP segment
Back-to-Back		77.9 μsecs	243.2 μsecs
Network Interfaces	Transmission	5.2 μsecs	121.1 μsecs
	Other	9.8 μsecs	11.7 μsecs
Total		92.9 μsecs	376.0 μsecs

Table 1: Non-processing related overhead removed from latency measurements.

Table 2 summarizes the processing of a single packet in the firewalls and routers for both Scout and Linux. The 1-byte numbers reveal that connection splicing achieves a considerable speedup. Most notably, the IP/FWD case is almost a factor of three faster than application-level forwarding. In terms of packets-per-second that can be processed by the firewall, this is an increase from 14,600 to 41,600. For large packets, the speedup is even greater—a factor of four. Eliminating the extra message copy and the checksum calculation required when transferring the message from one TCP connection to another accounts for the speedup.

Also note that in both the small and large message cases, the performance of the spliced connection is very close to the performance of the IP router configuration; the TCP header transformations amount to an extra 1.6 μsecs of processing. This suggests that any improvement made to IP router performance on this platform will be propagated to TCP forwarding. We expect that aggressive low-level optimizations, such as using polling instead of interrupts and employing a highly optimized classification algorithm, have the potential to improve IP routing performance (and hence TCP forwarding) by close to a factor of two. On a similar note, it would be interesting to wed connection splicing with hardware supported tag switching.

Comparing the Scout and the Linux numbers, we see that the 2-path case in Scout is slightly faster than the TIS firewall on Linux. IP router performance is approximately the same for the two systems. This indicates that other types of operating systems would also benefit from connection splicing. In a Linux implementation, the IP/FWD should perform close to that of the filtering IP forwarding—the updating of the TCP and IP headers would make it slightly slower. Keep in mind, however, that simple IP filtering does not permit a proxy that can sometimes operate in control mode.

Configuration		1-byte TCP segments		1460-byte TCP segments	
		Processing time (μsecs)	Speedup	Processing time (μsecs)	Speedup
Scout	2-path	68.5	–	101.1	–
	1-path	66.1	1.04	98.6	1.03
	FWD	39.0	1.76	39.5	2.56
	IP/FWD	24.0	2.85	24.0	4.21
	IP router	22.4	3.06	22.4	4.51
Linux	TIS Firewall	83.9	–	113.0	–
	Filtering IP router	27.5	3.05	29.0	3.90
	IP router	25.5	3.29	25.4	4.45

Table 2: Firewall and router processing per TCP segment.

5.2.2 Aggregate Throughput

The sustained throughput of a TCP forwarder is also a measure of its performance. The expectation is that the improved processing overhead of the optimized forwarders should allow them to support more concurrent TCP connections.

We measured the aggregate throughput of one, two, and three concurrent TCP connections over each configuration. Each TCP connection is between a client and a server from our test setup, such that each host supports only one TCP connection. The data unit transmitted by the client process was 1460 bytes. The aggregate throughput was obtained by adding the average throughput over the last 10 seconds of the individual connections. This was done when the throughput had reached a stable state. Not surprisingly, these measurements turned out to be bounded by the bandwidth of the 100 Mbit Ethernet, i.e., regardless of the number of TCP connections the aggregate throughput was close to 10 MB/s.

The more interesting question is how TCP forwarding behaves in the limit, that is, what bandwidth it can sustain. We can derive these numbers from the per-packet processing times presented in the previous section. For the 2-path and the IP/FWD configurations, we calculated the maximum throughput for different TCP acknowledgement patterns—either an acknowledgement is sent for every third, second, or single segment. For example, if an acknowledgement is sent for every third segment, the processing requirements for the data in the three segments would be three times the processing of a 1460-byte segment, plus the processing of a single empty segment. In our case, we have approximated the empty segment with a 1-byte segment.

The results are shown in Table 3. The 2-path TCP forwarder in our measurements is operating at almost maximum bandwidth of a 100 Mbit Ethernet, whereas the IP/FWD configuration is capable of supporting up to four times the bandwidth, corresponding to two full duplex OC-3 ATM connections.

Configuration		Message to acknowledgement ratio		
		3:1	2:1	1:1
Scout	2-path	11.7	10.8	8.6
	IP/FWD	45.6	40.6	30.4

Table 3: Estimated maximum throughput of firewall in MB/s

5.2.3 Cost of Splicing

The next question is how long it takes to splice (or unsplice) a forwarding path. As we have not yet implemented unsplicing, we focus on the cost of splicing two TCP connections. Our analysis has two parts. First, we establish the base processing overhead of splicing two TCP connections together. Second, we examine the end-to-end behavior of a TCP connection sending at maximum speed when the splicing is done.

To get the basic cost, we measured the time taken to splice two idle TCP connections. In this case, the measurements are free of any processing that might occur due to the draining of TCP buffers. In the test we continuously opened a TCP connection, waited 15 seconds and closed it again. The null proxy in the firewall optimizes the path 10 seconds after it is established. The numbers are the average time over 1000 such optimizations. Optimizing from TCP forwarding to FWD takes 25 μsecs on average. Adding the IP/FWD forwarding takes 94 μsecs on average. The higher cost of switching to IP/FWD is due to the fact that Scout requires a new path creation, whereas the FWD optimization is applied to the same path by doing code substitution.

As we concurrently forward new TCP segments and empty the buffers of the old segments, the cost of performing the optimization should be small even during high load. The more important question is whether or not the switch affects TCP's flow or congestion control algorithms. To see the effects of the switchover on a busy TCP connection, we performed the optimization 15 seconds into a throughput test. By tracing the sequence numbers of segments received at the server, we were unable to see any negative effects (Figure 11).

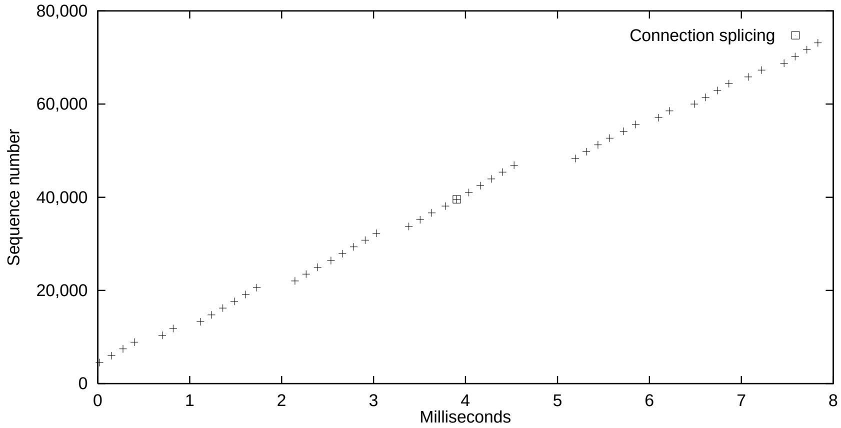

Figure 11: TCP Sequence Number Trace showing the effects of splicing

As moving to FWD forwarding reduces the processing with an average of 27.5 μsecs for small messages and 61.6 μsecs for large messages, it is always a good idea to switch to the FWD optimization, independent of how much data will flow over the spliced connection. Moving from FWD to IP/FWD reduces the processing by an extra 15 μsecs per packet, and thus, it will take six subsequent packets to make this optimization worthwhile.

5.3 Buffer Requirements

Buffer size is an issue for large-scale TCP forwarders. First, just having enough memory to accommodate thousands of TCP connections can be a problem, as each connection can easily require up to 256KB of buffering—two send buffers and two receive buffers of 64KB each. This translates to buffer requirements of 256MB for each 1,000 TCP connections to be supported. As the use of persistent TCP connections is becoming more widespread, thousands of connections per TCP forwarder is not uncommon. Splicing TCP connections together reduces the memory requirements of a TCP forwarder, since the forwarder is operating like an IP router and does not buffer segments.

Dynamic buffer allocation is another solution to this problem, but it requires processing to determine how much buffer to provide each connection. In this scenario, the TCP connections used for large data transfers are the most important. These TCP connections are the mostly likely candidates for splicing, thereby removing the buffer requirements all together. In other words, splicing can also make the administration of a TCP forwarder easier.

6 Related Work

In mobile computing, TCP forwarders are used to separate the TCP connection on a wireless link from that of a wired network [2]. This increases performance as the characteristics of the two types of networks are very different. As a mobile host moves around, it might sometimes connect directly to a wired network, in which case the TCP forwarder becomes superfluous and can be removed. This is done in the TACO system [7], where mobile hosts can—depending on what is required from their current type of network attachment—switch between having a TCP forwarder and not without destroying their TCP connections. The system differs from the one presented in this paper in two ways; it does not support filtering, and it uses interleaved connection establishment. This allows the TCP forwarder to be removed completely from the network path in the optimized case as no translation is necessary, but it at the same time limits the applicability of the solution. The lack of filtering makes it unsuitable for more advanced proxies such as firewalls.

Another research topic related to this paper is that of efficiently classifying packets [1, 6]. Of particular note are new algorithms to do fast routing table lookups based on variable length IP address prefixes [3, 10]. It is easy to imagine such techniques being extended to support fast IP filtering. Such an advance would be complementary to connection splicing, which can also exploit improved algorithms to determine to which path a particular packet belongs. Connection splicing is more general than IP filtering, however, since the proxy permits complex control operations.

7 Conclusion

This paper describes connection splicing, which can be applied to TCP forwarders to improve their performance. A performance study shows that an optimized TCP forwarder requires between one-half and one-quarter of the processing requirements of an unoptimized forwarder. The cost of the optimization varies according to how fast the buffers at the TCP forwarder can be emptied, but in most cases the cost is recovered within one to six packets. Furthermore, the optimization reduces the memory requirements of a TCP forwarder. The optimizations have been implemented in the Scout operating system, and it should be possible to get equivalent performance improvements in other systems.

Acknowledgments

We would also like to thank the other members of the Scout research group. This work supported in part by DARPA contracts DABT63-95-C-0075 and N66001-96-8518, and NSF grant NCR-9204393.

References

[1] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, 1994. ACM/USENIX.

[2] A. Bakre and B. Badrinath. Implementation and performance evaluation of indirect tcp. *IEEE Transactions on Computers*, 46(3), March 1997.

[3] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM '97 Symposium*, pages 3–14, Cannes, France, Sept. 1997. ACM.

[4] C. Brooks, M. Mazer, S. Meeks, and J. Miller. Application-Specific Proxy Servers as HTTP Stream Transducers. In *Electronic Proc. 4th Int. World Wide Web Conference "The Web Revolution"*, Boston, MA, Dec. 1995.

[5] M. Chatel. RFC 1919: Classical versus transparent IP proxies, Mar. 1996.

[6] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of SIGCOMM '96 Symposium*, pages 53–59, Stanford, CA, Aug. 1996. ACM.

[7] J. S. Hansen, T. Reich, B. Andersen, and E. Jul. Dynamic adaptation of network connections in mobile environments. *IEEE Internet Computing*, 2(1), January/February 1998.

[8] D. Mosberger and L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of OSDI '96*, October 1996.

[9] M. K. Ranum and F. M. Avolio. A toolkit and methods for Internet firewalls. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 37–44, Berkeley, CA, USA, Summer 1994. USENIX.

[10] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM '97 Symposium*, pages 25–38, Cannes, France, Sept. 1997. ACM.