TPOT: Translucent Proxying of TCP

Pablo Rodriguez[*]
EURECOM, France
rodrigue@eurecom.fr

Sandeep Sibal, Oliver Spatscheck
AT&T Labs – Research
{*sibal,spatsch*}*@research.att.com*

Abstract

Transparent Layer-4 proxies are being widely deployed in the current Internet to enable a vast variety of applications. These include Web proxy caching, transcoding, service differentiation, and load balancing. To ensure that all IP packets of an intercepted TCP connection are seen by the intercepting transparent proxy, they must sit at focal points in the network. *Translucent* Proxying of TCP (TPOT) overcomes this limitation by using TCP options and IP tunneling to ensure that all IP packets belonging to a TCP connection will traverse the proxy that intercepted the first packet. This guarantee allows the ad-hoc deployment of TPOT proxies *anywhere* within the network. No extra signaling support is required for its correct functioning. In addition to the advantages TPOT proxies offer at the application level, they also usually *improve* the throughput of intercepted TCP connections. In this paper we discuss the TPOT protocol, explain how it enables various applications, describe a prototype implementation, analyze its impact on the performance of TCP, and address scalability issues.

1 Introduction and Related Work

Transparent proxies are commonly used in solutions when an application is to be proxied in a manner that is completely oblivious to a client, without requiring any prior configuration. Recently, there has been a great deal of activity in the area of transparent proxies for Web caching. Several vendors in the area of Web proxy caching have announced dedicated Web proxy switches and appliances [1, 2, 8, 12].

In the simplest scenario, a transparent proxy intercepts all TCP connections that are routed through it. This may be refined by having the proxy intercept TCP connections destined only for specific ports (e.g., 80 for HTTP), or for a specific set of destination addresses. The proxy responds to the client request, masquerading as the remote web server. Scalability is achieved by partitioning client requests into separate hash buckets based on the destination address, effectively mapping web servers to multiple caches attached to the proxy.

In the event of a cache miss, the cache re-issues the request to the web server, and pipes the response it receives from the web server back to the client, keeping a copy for itself (assuming the response is cacheable). Note that, in general, this mechanism may be repeated, where a subsequent proxy along the path may intercept an earlier cache miss, and so on.

The proxy described above is often termed as a Layer-4 switch, or simply L-4 switch, since TCP is a Transport Layer protocol, which maps to Layer 4 in the OSI networking stack. In a variant of the above, the proxy/switch parses the HTTP request and extracts the URL and possibly other fields of the HTTP Request, before deciding what to do with the request. Since such a switch inspects the HTTP Request, which is an Application Layer or Layer 7 function, it is called an L-7 switch [2].

An acute problem that limits the use of transparent L-4 and L-7 Web proxies, is the need to have the proxy at a location that is guaranteed to see *all* the packets of the request [8]. Since routing in an IP network can lead to situations where multiple paths from client to server can have the lowest cost, packets of a connection may sometimes follow multiple paths. In such a situation a transparent proxy may see only a fraction of packets of the connection. Occasionally it is also possible that routes change mid-way through a TCP connection, due to routing updates in the underlying IP network. For these reasons transparent proxies are deployed exclusively at the edges or *focal* points within the network – such as gateways to/from single-homed clients or servers. This is not always the best place to deploy a cache. In general one would expect higher hit rates for objects cached deeper inside the network [9].

TPOT solves this problem by making an innovative use of TCP-OPTIONs and IP tunnels. A source initiating a TCP connection signals to potential proxies that it is TPOT-enabled by setting a TCP-OPTION within the SYN packet.

[*] He contributed to this work during an internship at AT&T.

A TPOT proxy, on seeing such a SYN packet, intercepts it. The ACK packet that it returns to the source carries the proxy's IP address stuffed within a TCP-OPTION. On receiving this ACK, the source sends the rest of the packets via the intercepting proxy over an IP tunnel. The protocol is discussed in detail in Section 2.

The above mechanism will work if the client is TPOT enabled. In a situation where the client is not TPOT enabled, we may still be able to use TPOT. As long as the client is single-homed, and has a proxy at a focal point, we can TPOT enable the connection by having the proxy behave like a regular transparent proxy on the side facing the client, but a TPOT (translucent) proxy on the side facing the server. Implementation of such a proxy is covered in Section 3.

The general idea of using TCP-OPTIONs as a signaling scheme for proxies is not new [20]. However combining this idea with IP tunneling to *pin down* the path of a TCP connection has not been proposed before to the best of our knowledge.

One alternative to TPOT is the use of Active Network techniques [31]. We believe that TPOT is a relatively lightweight solution that does not require an overhaul of existing IP networks. In addition, TPOT can be deployed incrementally in the current IP network, without disrupting other Internet traffic.

1.1 Applications of TPOT

In addition to allowing the placement of transparent Web proxy caches anywhere in the network, TPOT also enables newer architectures that employ Web proxy *networks*. In such architectures a proxy located along the path from the client to the server simply picks up the request and satisfies it from its own cache, or lets it pass through. This, in turn, may be picked up by another proxy further down the path. These incremental actions lead to the dynamic construction of spontaneous hierarchies rooted at the server. Such architectures require the placement of multiple proxies *within* the network, not just at their edges and gateways. Existing proposals [15, 21, 33] either need extra signaling, or they simply assume that all packets of the connection will pass through an intercepting proxy. Since TPOT explicitly provides this guarantee, implementing such architectures with TPOT is elegant and easy. With TPOT no extra signaling support or prior knowledge of neighboring proxies is required.

While the original motivation for TPOT was to enable Web proxy caching and Web proxy caching networks in an oblivious and ad-hoc fashion, the general idea of TPOT can also be applied to enable proxy-based services in a variety of other applications layered on top of TCP in an elegant and efficient fashion.

One such use is Transcoding. This refers to a broad class of problems that involve some sort of adaptation of content (e.g., [13, 23]), where content is transformed so as to increase transfer efficiency, or is distilled to suit the capabilities of the client. Another similar use is the notion of enabling a transformer tunnel [30] over a segment of the path within which data transfer is accomplished through some alternate technique that may be better suited to the specific properties of the link(s) traversed. Proposals that we know of in this space require one end-point to explicitly know of the existence of the other end-point – requiring either manual configuration or some external signaling/discovery protocol. TPOT can accomplish such functionality in a superior fashion. In TPOT an end-point noninvasively flags a connection, signifying that it can transform content – without actually performing any transformation. Only if and when a second TPOT proxy (capable of handling this transformation) sees this flag and notifies the first proxy of its existence, does the first proxy begin to transform the connection. Note that this does not require any additional handshake for this to operate correctly, since the TPOT mechanism plays out in concert with TCP's existing 3-way handshake.

Another use of TPOT is to enable the selection of specific applications for preferential treatment. Such service differentiation could be used to enable and enforce Q0S policies. One might also want to prioritize traffic belonging to an important set of clients, or a set of mission-critical servers.

1.2 Paper Overview

Section 2 describes the TPOT protocol. In addition to the basic version, a pipelined version of the protocol is also discussed. Pathological cases, extensions, and limitations are also studied.

Section 3 details a prototype implementation of TPOT in Scout [24]. We use this prototype in all our experiments.

We address the TCP level performance of TPOT in Section 4 using both theoretical analysis as well as experiments. Contrary to what we initially expected, TPOT typically *improves* the performance of TCP connections. This apparent counter-intuitive result has been observed before [3, 7, 16], though in somewhat different contexts. In [3] a modified TCP stack called *Indirect TCP* is employed for mobile hosts to combat problems of mobility and unreliability of wireless links. Results show that employing *Indirect TCP* outperforms regular TCP. In [16] similar improvements are reported for the case when TCP connections over a satellite link are split using a proxy. Finally, in [7], the authors discuss at length how TCP performance may be enhanced by using proxies for HFC networks. The notion of inserting proxies with the sole reason of enhancing performance has recently led to the coining of the term *Performance Enhancing Proxies* (PEP). An overview is provided in [5]. As we will see in Section 4, TPOT does indeed enhance performance, but unlike PEP, this is *not* the motivation behind TPOT.

Scalability is an important criterion if TPOT is to be

practically deployed. Section 5 discusses our approach to solving this problem using a technique that we call TPARTY, which employs a farm of servers that sit behind a front-end machine. The front-end machine only farms out requests to the army of TPOT machines that sit behind it. We show that our solution scales almost linearly with the number of TCP connections in the region of interest.

Finally Section 6 highlights our major contributions, discusses future work, and possible extensions to TPOT.

2 The TPOT Protocol

This section describes the operation of the basic and pipelined versions of the TPOT protocol. Pathological cases, extensions, and limitations are also studied. Before describing the operation of the TPOT protocol, we provide a brief background of IP and TCP which will help in better understanding TPOT. See [29] for a detailed discussion of TCP.

2.1 IP and TCP

Each IP packet typically contains an IP header and a TCP segment. The IP header contains the packet's source and destination IP address. The TCP segment itself contains a TCP header. The TCP header contains the source port and the destination port that the packet is intended for. This 4-tuple of the IP addresses and port numbers of the source and destination uniquely identify the TCP connection that the packet belongs to. In addition, the TCP header contains a flag that indicates whether it is a SYN packet, and also an ACK flag and sequence number that acknowledges the receipt of data from its peer. Finally, a TCP header might also contain TCP-OPTIONs that can be used for custom signaling.

In addition to the above basic format of an IP packet, an IP packet can also be encapsulated in another IP packet. At the source, this involves prefixing an IP header with the IP address of an intermediate tunnel point on an IP packet. On reaching the intermediate tunnel point, the IP header of the intermediary is stripped off. The (remaining) IP packet is then processed as usual. See RFC 2003 [27] for a longer discussion.

2.2 TPOT: Basic Version

In the basic version of TPOT a source S that intends to connect with destination D via TCP, as shown in Figure 1(a). Assume that the first (SYN) packet sent out by S to D reaches the intermediary TPOT proxy T. (S, Sp, D, Dp) is the notation that we use to describe a packet that is headed from S to D, and has Sp and Dp as the source and destination ports respectively.

To co-exist peacefully with other end-points that do not wish to talk TPOT, we use a special TCP-OPTION "TPOT," that a source uses to explicitly indicate to TPOT proxies within the network, such as T, that they are interested in using the TPOT mechanism. If T does not see this option, it will take no action, and simply forwards the packet on to D on its fast-path. If T sees a SYN packet that has the TCP-OPTION "TPOT" set, it responds to S with a SYN-ACK that encodes its own IP address T in the TCP-OPTION field. On receiving this packet, S must then send the remaining packets of that TCP connection, IP tunneled to T. From an implementation standpoint this would imply adding another 20 byte IP header with T's IP address as destination address to all packets that S sends out for that TCP connection. Since this additional header is removed on the next TPOT proxy, the total overhead is limited to 20 bytes regardless of the number of TPOT proxies intercepting the connection from the source to the final destination. This overhead can be further reduced by IP header compression [10, 18].

For applications such as Web Caching where T may be able to satisfy a request from S, the response is simply served from one or more caches attached to T. In the case of a "cache miss" or for other applications where T might connect to D after inspecting some data, T communicates with the destination D as shown in Figure 1(a). Note that the proxy T sets the TCP-OPTION "TPOT" in its SYN to D to allow possibly another TPOT proxy along the way to again proxy the connection. Note that Figure 1 only shows the single proxy scenario.

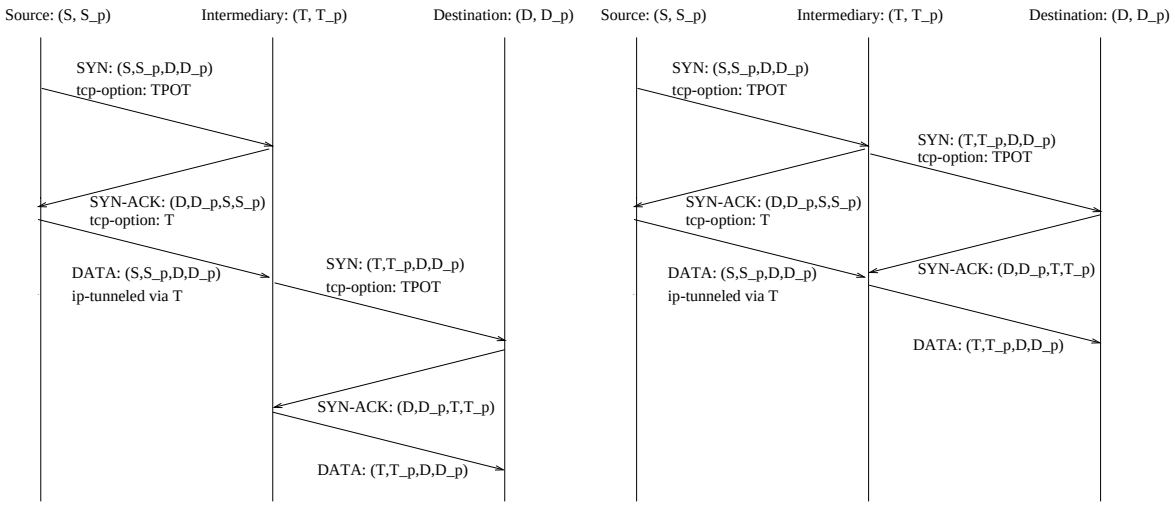
2.3 TPOT: Pipelined Version

In certain situations one can do *better* that the basic version of the TPOT protocol. It is possible for T to pipeline the handshake by sending out the SYN to D immediately after receiving the SYN from S. This *pipelined* version of TPOT is depicted in figure 1(b).

The degree of pipelining depends on the objective of the proxying mechanism. In the case of an L-4 proxy for Web Caching, the initial SYN contains the destination IP address and port number. Since L-4 proxies do not inspect the content, no further information is needed from the connection before deciding a course of action. In such a situation a SYN can be sent out by T to D almost immediately after T received a SYN from S, as shown in Figure 1(b). In the case of L-7 switching, however, the proxy T would need to inspect the HTTP Request (or at a minimum the URL in the Request). Since this is typically not sent with the SYN, a SYN sent out to D can only happen after the first ACK is received by T from S. This is consistent with Figure 1.

2.4 Pathological Cases

While the typical operation of TPOT appears correct, we are aware of two pathological cases that also need to be addressed.



(a) Basic Version

(b) Pipelined Version

Figure 1: The TPOT protocol

1. In a situation when a SYN is retransmitted by S, it is possible that the retransmitted SYN is intercepted by T, while the first SYN is not – or vice versa. In such a situation, S may receive SYN-ACKs from both D as well as T. In such a situation S simply ignores the second SYN-ACK, by sending a RST to the source of the second SYN-ACK.

2. Yet another scenario, is a simultaneous open from S to D and vice-versa, that uses the same port number. Further T intercepts only one of the SYNs. This is a situation that does not arise in the client-server applications which we envision for TPOT. Since S can turn on TPOT for only those TCP connections for which TPOT is appropriate, this scenario is not a cause for concern.

2.5 Extensions

As a further sophistication to the TPOT protocol it is possible for multiple proxied TCP connections at a client or proxy that terminate at the same (next-hop) proxy, to integrate their congestion control and loss recovery at the TCP level. Mechanisms such as *TCP-Int* proposed in [4] can be employed in TPOT as well. Since the primary focus of TPOT, and this paper, is to enable proxy services on-the-fly, rather than enhance performance we do not discuss this further. The interested reader is directed to [4] and [32] for such a discussion.

Note that an alternative approach is to multiplex several TCP connections onto a single TCP connection. This is generally more complex as it requires the demarcation of the multiple data-streams, so that they may be sensibly demultiplexed at the other end. Proposals such as P-HTTP [22] and MUX [14], which use this approach, may also be built into TPOT.

2.6 Limitations

As shown in Figure 1 the TCP connection that the intermediate proxy T initiates to the destination D will carry T's IP address. This defeats any IP-based access-control or authentication that D may use. Note that this limitation is not germane to TPOT, and in general, is true of any transparent or explicit proxying mechanism.

In a situation where the entire payload of an IP packet is encrypted, as is the case with IPsec, TPOT will simply not be enabled. This does not break TPOT, it simply restricts its use.

The *purist* may also object to TPOT breaking the semantics of TCP, since in TPOT a proxy T in general interacts with S, in a fashion that is asynchronous with its interaction with D. While it is possible to construct a version of TPOT that preserves the semantics of TCP, we do not pursue it here. In defense, we point to several applications that are prolific on the Internet today (such as firewalls) that are just as promiscuous as TPOT.

3 Implementing TPOT in Scout

TPOT can be implemented in any operating system. This section describes an implementation in an OS designed specifically to support communication: Scout [24]. While

the primary purpose of this section is to flesh out some of the details any implementation would have to address, it has a secondary objective of illustrating how a technique like TPOT can be naturally realized in an operating system designed around communication-oriented abstractions. Many overheads and latency penalties incurred by proxies on general purpose operating systems like Linux, BSD or WindowsNT can be avoided by such an operating system.

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 they move 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 they move through the system, and (2) it represents the entity that is scheduled for execution.

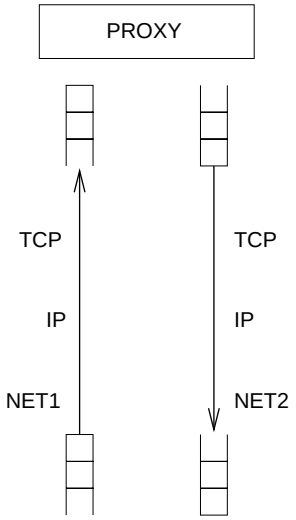


Figure 2: TCP proxy in two Scout paths.

The path abstraction lends itself to a natural implementation of TCP proxies. Figure 2 schematically depicts a naive implementation of a proxy 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]. As a first approximation, the configuration of Scout shown in Figure 2 represents the implementation one would expect in a traditional OS.

The two-path configuration shown in Figure 2 has suboptimal performance because it requires the hand-off of each incoming segment from the first path to the proxy, and then again from the proxy to the second path. In Scout, the entire device-to-device data flow can be encapsulated in a single path as shown in Figure 3. This is the implementation of choice for TCP proxying within Scout. The TPOT protocol is then implemented within this base TCP implementation.

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

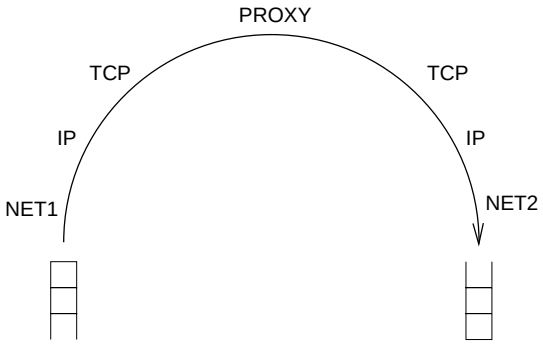


Figure 3: TCP proxy implemented in a single Scout path.

Figure 4 illustrates two configurations used in the experiments. The configuration on the left implements TPOT only in one half of the TCP path. This configuration is used as a client-side proxy for TPOT-enabling those clients that do not implement TPOT themselves. The side facing the client behaves like a regular transparent proxy. As discussed earlier, this solution will work only if this proxy is at a focal point of the network with respect to the client. The side facing the destination behaves like a full-fledged TPOT proxy, issuing TCP SYN requests with the TPOT option set.

The configuration on the right is used to pick up TCP SYN requests which have the TPOT option set. It may then terminate such a TCP connection and establish an IP tunnel back to the initiator of the TCP connection using the TPOT protocol. It then initiates a second TCP connection towards the original destination. In both configurations the connection establishment for both the TCP connections is performed in parallel, as per the pipelined version of TPOT (see Figure 1(b)).

The IP tunnel modules "IP in IP" shown in Figure 4 attach and remove IP tunnel headers [27]. IP tunnel headers are added to all IP packets sent after a SYN or a SYN ACK with a TPOT option set, has been received. The inner IP module spoofs for the original destination of the TCP connection. The outer IP module uses the real IP addresses of the originator of the TCP connection and the TPOT proxy which terminated the connection.

Scout's TCP implementation is derived from the TCP implementation in the x-kernel [17], which in turn is derived from a BSD TCP implementation. To match the TCP implementation in the Linux client and server for the experiments in the following section, (in addition to TPOT) we enabled the MSS, and SACK options in Scout. The receive

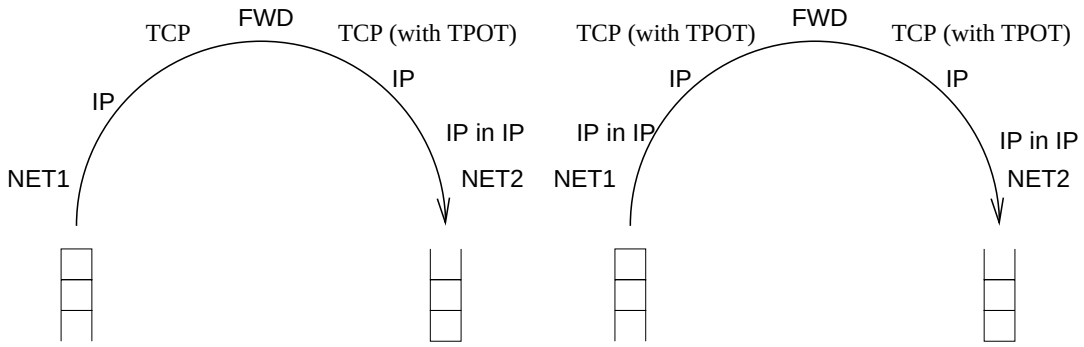


Figure 4: TPOT implementation in Scout.

buffer is also set to 32KByte to match the values used by the Linux client and server.

4 Performance Measurements

This section analyzes the TCP performance of TPOT based on actual measurements in lab setups using prototype TPOT proxies. Wherever relevant, we compare the observed performance with expected values suggested by theoretical results on the performance of idealized TCP. In our experiments we use the Reno flavor of TCP [29], which is generally considered to be the most popular implementation on the Internet today. We expect our observations to largely hold for other flavors of TCP, though it is quite possible that flavors of TCP such as TCP-Vegas [6], which have different congestion detection and avoidance techniques, may yield somewhat different numbers.

The primary focus of the following experiments is to evaluate the performance benefits and penalties in the presence of realistic Round-trip-times (RTTs) and packet losses, when one or more TPOT machines intercepts a TCP connection. For these experiments the TPOT machines are not overloaded. Section 5 discusses overload situations, and techniques of scaling TPOT to combat this.

For our experiments we tested the pipelined version of TPOT (See Section 2). In the worst case the basic version would incur an additional delay of half a round-trip-time. Aside from this, the two versions of TPOT yield similar results.

4.1 Setup

All hosts used in our experiment are at least 200 MHz Pentium II workstations with 256KB cache, 32MB RAM, and 3COM 3x59 32-bit PCI 10/100 Mb/s adapters. The first TPOT machine runs the transparent proxy version of TPOT, while the second TPOT machine runs the interior TPOT version which are described in the previous section. The clients and servers are Linux 2.2.12 machines. The physical configuration of our test setup is shown in Figure 5. The client is connected with a 10Mbit hub to the first TPOT machine. The first TPOT machine is connected by another 10Mbit Hub to the second TPOT machine. The second TPOT machine is in turn connected by a 10Mbit hub to the server.

Figure 5: Test Setup

The TPOT machines either operate as TPOT proxies or as simple routers. If they operate as TPOT proxies the first TPOT machine enables the TPOT protocol and data is subsequently tunneled between the TPOT machines. Delays and losses are added in the device driver code of each TPOT device. The granularity of the delay queue is 1 ms. For throughput measurements TTCP is used to measure the throughput on the receiver. TTCP transfers a specified amount of data from the client to the server. After all the data has been transfered, the connection is closed. The results reported for each experiment, are averaged over ten runs.

The Linux TCP code implements the Timestamp option which is not supported by Scout. We believe that the impact of this shortcoming is minor in our test environments which by design have low RTT variances. Despite that fact that both Linux and Scout advertise the SACK option during initial handshake, tcpdump traces show that SACK was not used during the data transfer phase of the TCP connections in any of the experiments.

4.2 Impact of RTT

To measure the impact of the RTT, we introduced delay into the output queue of the Ethernet devices on the TPOT machines. In one set of experiments the TPOT machines work exclusively as routers, and in the second set exclusively as TPOT proxies. In the second experiment the distribution of the added delay, is either equally distributed

over all links, or is concentrated on a single link between the two TPOT machines.

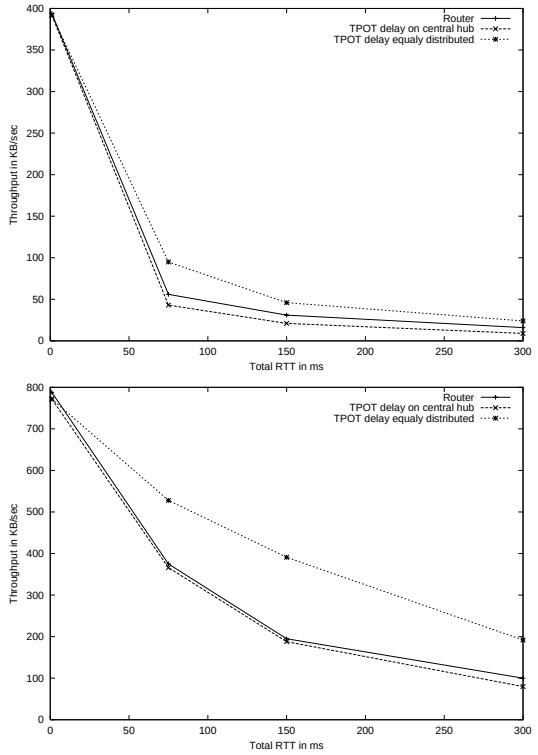


Figure 6: Throughput for different RTTs for 10KB (top) and 10MB (bottom) document sizes.

Figure 6 shows the throughput for RTTs from 1-300 ms for 10KB and 10MB document sizes. Smaller documents are not measured since the connection establishment time dominates the experiment. The impact on small documents is discussed in section 4.5. The results show that if the entire RTT is concentrated at the single link between the two TPOT machines, the throughput is on average 24% worse for 10KB documents and 6% worse for 10MB documents. This is not surprising since the TPOT machines need to perform additional processing during connection setup which gets amortized over the lifetime of the connection, in addition to the processing for each packet.

On the other hand in the case when the RTT is equally distributed over the links, we find that TPOT *improves* the overall throughput. For example, the TPOT throughput for a 300ms RTT and 10MB documents is 92% better than the routed throughput for the same RTT.

Theoretical Analysis

To better understand this phenomenon we turn our attention to results in the literature that analyze the performance of TCP using idealized models. Note that this section is intended as a theoretical backing for our study, and is not intended as a comprehensive or formal analysis of TCP.

In [11] the authors provide a rough sketch for the throughput of an idealized TCP connection in the congestion avoidance phase. A more rigorous derivation of this and a few other results may be found in [25]. The authors of [26] model TCP throughput in a more comprehensive fashion taking into account TCP timeouts as well. We use the terminology of [26] in what follows.

Let p_i and RTT_i be the packet loss and RTT on link i, and $B_i(p_i)$ be the corresponding throughput in *packets per second*. Also, let W_{max} be the maximum advertised window size. Let the constant b, be the number of packets acknowledged by each ACK. Then in steady-state, as per [26]:

$$B_i(p_i) \approx min\left(\frac{W_{max}}{RTT_i}, \frac{1}{RTT_i}\sqrt{\frac{3}{2 \cdot b \cdot p_i}}\right) \qquad (1)$$

Note that the above equation ignores timeouts. Including timeouts does not change the *nature* of the analysis that follows. A detailed discussion is beyond the scope of this paper.

For connections with a high RTT the advertised window size W_{max} becomes the bottleneck, so that the above equation reduces to:

$$B_i(p_i) \approx \frac{W_{max}}{RTT_i} \qquad (2)$$

If we assume (in the optimal case) that the buffers along the intermediate proxies are rarely starved, the maximum end-to-end throughput B^* may be expressed as:

$$\begin{aligned} B^* &\approx min_i\left(B_i(p_i)\right) \\ &\approx min_i\left(\frac{W_{max}}{RTT_i}\right) \\ &\approx \frac{W_{max}}{max_i(RTT_i)} \end{aligned}$$

In other words, the overall throughput is primarily determined by the link i with the longest RTT_i. Since the total round trip time $RTT = \sum_i RTT_i$ is conserved, creating N equal length links by inserting proxies, can potentially (at most) multiply the end-to-end throughput by a factor of N. In our experiments, as shown in Figure 6, the throughput improvement however is more modest than this upper bound.

While a detailed analysis of the exact multiplicative factor is beyond the scope of this paper, it is obvious that an improvement in steady-state TCP throughput is to be expected. In addition, it is also conceivable that the TCP implementations within each entity may create peculiar interactions due to the way scheduling is done between sending data on the outgoing TCP connection, and processing acknowledgments on the incoming connection.

TCP's scaled window option [19], which was not used in these experiments, would improve the throughput of the proxied and the un-proxied connections by an equal

amount. If the scaled window, W_{max}, becomes so large that it is no longer the bottleneck, the send window will become the limiting factor. This case is discussed in the next section.

4.3 Impact of Loss

While the advertised window size was the determining factor of B^* for high RTT connections in the previous experiment, the goal of this experiment is to demonstrate that TPOT also performs better if the sender's congestion window and not the receiver's advertised window limits throughput. To study this scenario we randomly drop packets in an uniform and independent fashion in the Ethernet device driver of the TPOT machines. In this experiment no artificial delay is introduced. This results in an RTT of 1ms between the client and server due to the real delay on the Ethernet and TPOT machines. The idea here is to simulate packet losses either due to lossy links or packet loss due to buffer overflows along the path. Again we measure the performance of end to end TCP using the TPOT machines configured exclusively as routers, or as TPOT proxies. In the case where they are configured as proxies the loss is either equally distributed between the links, or is concentrated on the link between the two TPOT machines.

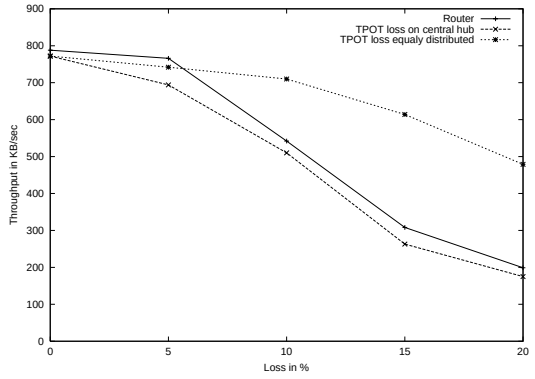


Figure 7: Throughput for different drop rates.

Figure 7 depicts the results of this experiment for 10MB document sizes for different loss rates. The experiment for 10KB is not reported since the results were highly variant. This is because of the timeout behavior of TCP SYN packets, and the fact that the the total number of packets transfered is low.

Figure 7 shows that the Router version is slightly better than the TPOT proxy version with packet loss concentrated on the central link. We believe this is due to the overhead involved in introucing TPOT proxies. However when the packet loss is equally distributed, The TPOT proxy version outperforms the Router version by far. Note that this shows up only for throughput values below 600KBps, since above this, the 10Mbps Ethernet dominates the picture.

Theoretical Analysis

In this situation, the RTT_i of each link i is the same. Let us denote this by RTT_x. When the throughput is not dominated by W_{max}, Equation 1 reduces to:

$$B_i(p_i) \approx \frac{1}{RTT_x}\sqrt{\frac{3}{2 \cdot b \cdot p_i}} \tag{3}$$

The upper bound on the end-to-end throughput B^* may then be expressed as:

$$\begin{aligned} B^* &\approx min_i\left(B_i(p_i)\right) \\ &\approx min_i\left(\frac{1}{RTT_x}\sqrt{\frac{3}{2 \cdot b \cdot p_i}}\right) \\ &\approx \sqrt{\frac{3}{2 \cdot b \cdot RTT_x}} \cdot \frac{1}{\sqrt{max_i(p_i)}} \end{aligned}$$

B^* is thus determined by the most lossy link. Note that in this case the overall loss probability is conserved, so that, $p = 1 - \Pi_i(1 - p_i)$. The results of the experiments roughly corroborate this. For the 10MB document size, we see that after the throughput drops below the Ethernet saturation point, the equally distributed packet loss case, outperforms the router case significantly – in fact slightly more than the theoretically expected value of $\sqrt{3}$. Again, a detailed theoretical analysis is beyond the scope of this paper.

4.4 A Simple Case Study

The previous two experiments give an idea of the behavior of TCP proxies in the worst and best case scenarios in terms of the distribution of RTTs *or* packet loss rate. The following three experiments focus on studying the impact of TPOT when deployed in a realistic setting in the Internet with varying RTTs *and* packet loss rate. The topology of the setup is similar to the one used earlier. See Figure 5.

The first experiment (I) has one TPOT machines configured close to the server and one close to the client. All losses and delays are in the middle of the network. This simulates the case where the transparent proxies/switches are on the local network of the server and the client. In fact, this is the worst case, as can be deduced from the experiments above. The second case (II) uses the same setup, but redistributes the RTT and packet drop rate, so as to better simulate a user dialing into an ISP. The third setup (III) moves the transparent proxy on the server-side further into the backbone, resulting in even better performance. The total simulated latency is always 250ms and the total simulated drop rate is 9.74%.

The above table shows the individual RTTs and drop rates between the client and the first proxy (link 1), the first proxy and the second proxy/switch (link 2), and the second proxy/switch and the server for the three different setups. As mentioned earlier, the general setup shown in Figure 5 was used. In all cases, the throughput of a TTCP transfer

Case	Link 1 (RTT/drop)	Link 2 (RTT/drop)	Link 3 (RTT/drop)
I	1ms/0%	248ms/9.74%	1ms/0%
II	110ms/3%	139ms/6.96%	1ms/0%
III	110ms/3%	70ms/1%	70ms/6%

Table 1: RTT and packet drop rate distributions for the experiments.

of size 100KB was measured from the client to the server. The TPOT machines, as in the previous experiments, were either used as routers or as TPOT proxies. The RTT and packet drop rate distribution was the same in either case.

Case	TPOT throughput	Routed throughput
I	26 kBps	24 kBps
II	53 kBps	24 kBps
III	79 kBps	24 kBps

Table 2: 100KB transfer for case I-III with and without TPOT.

Table 2 shows the results of the experiment. Not surprisingly, the throughput remains the same for all cases, when the TPOT machines are configured as routers. It does not make any difference where the data is lost or where RTT delays are introduced as long as the end to end RTT and loss are equal. Also, not surprisingly, the throughput of the proxied TCP increases by more than a factor of three due to the reduction of the individual TCP connections drop rate and RTT. This case study also shows that additional benefits can be derived from the TPOT architecture. Case I and II can be implemented without TPOT since all proxies can be placed on focal points in the network. However, case III is possible only if the connection is TPOT enabled, since in Case III the proxy/switch is in the middle of the network.

4.5 Small Data Transfers

Another important question is how TPOT effects small data transfers. The previous experiments have shown that throughput increases when TPOT proxies are used. However, for small files, the connection establishment overhead dominates the overall performance, and sustained throughput rates become irrelevant. To measure the effect of TPOT on small file transfers, the setup in the previous experiments was used. However, instead of TTCP which transfers data in one direction, we used a TCP ping test which returns the data back to the sender simulating a HTTP Request followed by a short HTTP Response. The total time from before the open system call to after the close system call on the client side of the connection was measured using the hardware cycle counter of the Pentium II processor.

Table 3 shows the results for three transfer sizes and two different values for RTT. The delay was equally distributed between all links. As expected, the overall transfer time increases for very small RTTs and file sizes. In the 1ms 1Byte case the transfer time increased by 1.5 ms or 55%. However as the transfer size increases, the additional processing overhead of establishing two TCP connections on the proxy amortizes itself over the duration of the connection.

RTT	Transfer size	TPOT Delay	Routed Delay
1	1B	4.2 ms	2.7 ms
1	1KB	8.5 ms	7.9 ms
1	10KB	33.9 ms	33.5 ms
100	1B	137.3 ms	206.2 ms
100	1KB	141.4 ms	210.7 ms
100	10KB	374.3 ms	636.1 ms

Table 3: Transfer time for a short Request-Response data transfer.

For high values of RTT, TPOT actually reduces the transfer time. The 1Byte transfer is 69ms or 50% faster than the same transfer when the TPOT machines are configured as routers. This is because the connection establishment handshake between the individual TPOT machines is performed concurrently, and each individual TCP connection has only 1/3 of the total RTT between the client and server.

5 Scalability Issues

In the previous section we saw that the throughput and latency of TCP connections can be improved, under the tacit assumption that the TPOT machines are not in overload. This section discusses how we may tackle situations when a single TPOT proxy is unable to handle its assigned load. Fortunately, functionality within a TPOT proxy can be easily parallelized. This allows the scaling of TPOT connections to support a potentially large number of TCP connections. This section addresses the issues related with scaling TPOT, and provides measurements on the processing overhead introduced on the TPOT machine due to a connection. These measurements may be used to estimate the cost and size of a *TPARTY* – which is essentially a farm of TPOT proxies front-ended by a modified high-bandwidth router.

5.1 Processing Overhead

One important factor for scalability is the number of CPU cycles needed to TPOT a TCP connection. Unfortunately this number depends on many factors. The CPU, the operating system, the particular TCP implementation and the loss rate of the link are only a few. Therefore, it is infeasible to answer this question in general and the results of this sections should only be used to gather a general understanding of the performance required.

The results presented here were measured on a 351MHz PentiumII running Scout. Scout provides a unique feature [28] which allows us to measure the CPU consumption of all computation performed for a single path in kernel or user space, within the accuracy of the CPU hardware cycle counter. Using this feature, the number of cycles to transfer different data sizes using TTCP and the TCP ping test, described in the previous section, are measured.

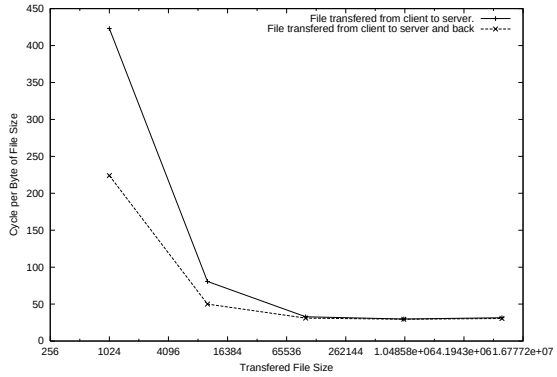


Figure 8: Processing overhead on a 351MHz PentiumII – per Byte of data transfered.

To proxy a single TCP connection, including the open and close handshakes of both the incoming and outgoing TCP connections, TPOT requires on average 431.568 KiloCycles. The cycles/byte for larger data transfers is shown in Figure 8. The cycles per byte shown includes the cycles needed for the handshake. Figure 8 shows that for data transfers over 100KB, the processing needed is dominated by the per byte cost (which are on the order of 30 cycles per byte), and is not influenced in any significant way by the handshake. This high number seems to indicate unnecessary copying.

5.2 TPARTY

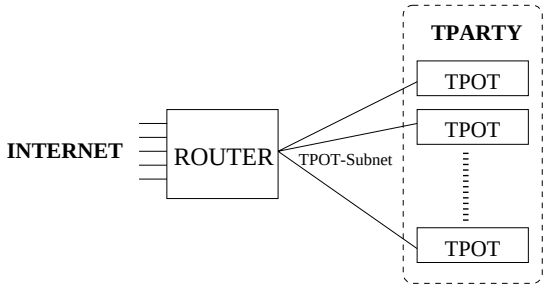


Figure 9: TPARTY: parallelizing TPOT proxies.

TPARTY can be used to scale TPOT in cases when the load cannot be supported by a single TPOT machine. The basic idea is depicted in Figure 9. TPARTY uses a farm of TPOT proxies front-ended by a modified router. In addition to routing, the router forwards TCP SYN packets for certain TCP port numbers which have the TPOT option enabled (and are not received on the line card connected to the TPOT subnet) towards one of the TPOT machines. The router might forward the TPOT-enabled SYNs – and therefore the handling of TCP connections – in round-robin fashion, or might use feedback information on the load at each of the TPOT machines to make a more intelligent decision.

When a TPOT-enabled SYN arrives at the TPOT machine, the TPOT machine decides if it can handle an additional request. If it cannot handle the request, the SYN is sent back to the router and the packet is routed as usual to the final destination. In this case the connection will not be proxied. If the TPOT machine has enough resources to deal with the connection, the proxy terminates the connection as described in the TPOT protocol using the IP address of the individual TPOT machine as proxy address. In either case, all subsequent packets on the router are routed as plain IP packets. Therefore, the additional processing on the router is limited to detecting and forwarding of TPOT enabled SYN packets.

Using the measurements from Section 5.1, a TPARTY for an OC-3 link would require 155Mbit/8*30 Cycles per byte (roughly 581 MegaCycles) to sustain the link bandwidth. In addition an extra 1000*431568 = 431 MegaCycles would be required for the open and close of 1000 connections per seconds. Therefore OC-3 speeds would require on the order of two to three 500MHz Pentium III machines.

5.3 Buffer Size

The buffering required on the TPOT machines is another cost of performing proxying in the middle of the network. Each proxy requires a send and receive buffer to terminate the TCP connection. In theory all buffering will happen before the worst link (some combination of the highest RTT and highest packet drop rate), and little or no buffering would be required thereafter. In general, buffering on the order of the advertised window by the remote receiver of the TCP connection on the proxy is required. More studies need to be conducted for estimating the buffer requirements for the TPOT proxy. Buffer sharing across multiple connections may help decrease the total buffer space consumed by a TPOT proxy.

5.4 Port Numbers

Another scaling problem arises from the fact that demultiplexing of TCP connections is based on source and destination IP addresses and port numbers. The destination address and port number are usually fixed. The source address seen by the final destination will be the address of the

last TPOT proxy. This limits the last proxy to open at most 64K connections (range of the port number space) to a single server. Since ports may time out substantially after a connection is closed, the number of active connections the last TPOT proxy can handle is far lower. The exact number depends on the TCP implementation, but clearly this is a potential bottleneck.

One solution to this problem is to multiplex several TCP connections onto a single TCP connection. This would allow the reuse of an existing outgoing connection from the last TPOT proxy to the final destination. This is possible if the destination supports Persistent-HTTP [22]. Alternate techniques such as MUX [14] may also be used.

Another solution is to multi-home the TPOT machine, so that a single TPOT machine may be assigned multiple IP addresses. Since the bound of 64K connections holds for a single IP address, multi-homing multiplies this by the number of IP addresses assigned to the TPOT machine.

6 Conclusions and Future Work

In this paper we evaluated the benefits of using TPOT – a *translucent* mechanism for proxying TCP connections. In general, transparent proxies do not always see all the packets of a TCP connection, unless they are placed at focal points within the network. TPOT proxies do not suffer from this limitation because of a novel way in which TCP-OPTIONs and IP tunneling are used to *pin down* TCP connections.

Web proxy caches built using TPOT thus have the freedom of being placed *anywhere* in the network, which in turn enables new architectures such as spontaneous Web proxy *networks*, where proxy caching hierarchies are *dynamically* constructed. In addition, transcoding and several other applications can benefit from TPOT. To test our ideas we implemented a prototype TPOT proxy in Scout. Much of this paper was focused on addressing TCP performance and scalability concerns associated with deploying TPOT in real networks. Our preliminary assessment indicates that TPOT scales exceptionally well when it is configured as a TPARTY – which is a farm of TPOT proxies with a modified router as a front-end. We also discovered that proxying connections actually *improves* the overall performance of the connection at the TCP level. Theoretical analysis backed our measurements. Our measurements also indicate that the overhead due to protocol processing at the proxy is usually more than compensated by the improved throughput seen at the TCP level.

We believe the results of our work indicate that TPOT is viable, and can be practically deployed in a high-speed network to enable a variety of applications. We are currently investigating its use in building *zero-maintenance* proxy caching networks for the Web.

Acknowledgments

We thank Misha Rabinovich, Adam Buchsbaum, and Fred Douglis of AT&T Labs – Research, for their comments on an earlier draft of this paper.

References

[1] Alteon Networks. Webworking: Networking with the Web in mind. Technical report, San Jose, CA, USA, May 1999.

[2] ArrowPoint Communications. What is Web Switching? Technical report, Westford, MA, USA, 1999.

[3] A. Bakre and B. R. Badrinath. Indirect TCP for mobile hosts. In *International Conference on Distributed Computing Systems (ICDCS)*, May 1995.

[4] H. Balakrishnan, V. Padmanabhan, S. Seshan, R. H. Katz, and M. Stemm. TCP behavior of a busy internet server: Analysis and improvements. In *Proceedings of IEEE INFOCOM*, San Francisco, CA, March 1998.

[5] J. Border, M. Kojo, J. Griner, and G. Montenegro. Performance enhancing proxies. Technical report, IETF, June 1999. Internet Draft draft-ietf-pilc-pep-00.txt.

[6] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM Conference*, London, England, 1994.

[7] R. Cohen and S. Ramanathan. TCP for high performance in hybrid fiber coaxial broad-band access networks. *IEEE/ACM Transactions on Networking*, 6(1):15–29, Feb. 1998.

[8] P. Danzig and K. L. Swartz. Transparent, scalable, fail-safe Web caching. Technical report, Network Appliance, Santa Clara, CA, USA, 1999.

[9] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of the ACM SIGCOMM Conference*, September 1993.

[10] M. Degermark, B. Nordgren, and S. Pink. RFC 2507: IP header compression, Feb. 1999. Status: PROPOSED STANDARD.

[11] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control on the Internet. *IEEE/ACM Transactions on Networking*, 1999.

[12] Foundry Networks. Transparent cache switching primer. Technical report, Sunnyvale, CA, USA, 1999.

[13] A. Fox and E. A. Brewer. Reducing WWW latency and bandwidth requirements by real-time distillation. *Computer Networks and ISDN Systems*, 28(7–11):1445–1456, May 1996.

[14] J. Gettys. Mux protocol specification. Technical report, W3C, October 1996. wd-mux-961023.

[15] A. Heddaya and S. Mirdad. Webwave: Globally balanced fully distributed caching of hot published documents. In *International Conference on Distributed Computing Systems (ICDCS)*, Baltimore, USA, May 1997.

[16] T. R. Henderson and R. H. Katz. Transport Protocols for Internet-compatible Satellite Networks. *IEEE Journal on Selected Areas in Communications*, 1999. To appear.

[17] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[18] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, Feb. 1990. Status: PROPOSED STANDARD.

[19] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, May 1992.

[20] B. Knutsson. Proxies and signalling. *Extendable Router Workshop*, August 1999.

[21] P. Krishnan, D. Raz, and Y. Shavitt. Transparent en-route cache location in regular networks. In *DIMACS Workshop on Robust Communication Networks: Interconnection and Survivability*, DIMACS book series, New Brunswick, NJ, USA, November 1998.

[22] J. C. Mogul. The case for persistent-connection HTTP. In *Proceedings of the SIGCOMM'95 conference*, Cambridge, MA, August 1995.

[23] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM Conference*, September 1997.

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

[25] T. J. Ott, J. H. B. Kemperman, and M. Mathis. The stationary behavior of ideal TCP congestion avoidance. ftp://ftp.bellcore.com/pub/tjo/TCPwindow.ps, 1996.

[26] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empricial validation. In *Proceedings of the ACM SIGCOMM Conference*, Vancouver, British Columbia, Canada, September 1998.

[27] C. Perkins. RFC 2003: IP encapsulation within IP, Oct. 1996. Status: PROPOSED STANDARD.

[28] O. Spatscheck and L. L. Peterson. Definding against denial of service attacks in Scout. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 59–73, New Orleans, Louisiana, Feb. 199. USENIX Association.

[29] W. R. Stevens. *TCP/IP Illustrated, The Protocols*, volume 1. Addison-Wesley, 1994.

[30] P. Sudame and B. R. Badrinath. Transformer tunnels: A framework for providing route-specific adaptations. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, USA, June 1998.

[31] D. L. Tennenhouse, J. M. Smith, D. Sincoskie, D. J. Wetherhall, and G. J. Minden. A survey of Active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[32] J. Touch. TCP control block interdependence. Technical report, IETF, April 1997. Interent RFC 2140.

[33] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web caching. In *NLANR Web Caching Workshop*, June 1997.