Upgrading Transport Protocols using Untrusted Mobile Code

Parveen Patel† ppatel@cs.utah.edu
Andrew Whitaker‡ andrew@cs.washington.edu
David Wetherall‡ djw@cs.washington.edu
Jay Lepreau† lepreau@cs.utah.edu
Tim Stack† stack@cs.utah.edu

†University of Utah ‡University of Washington

ABSTRACT

In this paper, we present STP, a system in which communicating end hosts use untrusted mobile code to remotely upgrade each other with the transport protocols that they use to communicate. New transport protocols are written in a type-safe version of C, distributed out-of-band, and run in-kernel. Communicating peers select a transport protocol to use as part of a TCP-like connection setup handshake that is backwards-compatible with TCP and incurs minimum connection setup latency. New transports can be invoked by unmodified applications. By providing a late binding of protocols to hosts, STP removes many of the delays and constraints that are otherwise commonplace when upgrading the transport protocols deployed on the Internet. STP is simultaneously able to provide a high level of security and performance. It allows each host to protect itself from untrusted transport code and to ensure that this code does not harm other network users by sending significantly faster than a compliant TCP. It runs untrusted code with low enough overhead that new transport protocols can sustain near gigabit rates on commodity hardware. We believe that these properties, plus compatibility with existing applications and transports, complete the features that are needed to make STP useful in practice.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management; D.4.6 [**Operating Systems**]: Security and Protection; C.2.2 [**Network Protocols**]: Protocol architecture

General Terms

Design, Implementation, Deployment

Keywords

Transport Protocols, TCP-friendliness, Untrusted Mobile Code

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

Upgrading transport protocols in the global Internet is hard. TCP extensions [54, 41, 50, 13] and new protocols, such as SCTP [56] and DCCP [37], generally require both communication endpoints to be upgraded before they provide the benefit of new and improved functionality to users. This is presently done by independent operating system upgrades at individual endpoints. Furthermore, new protocol options need to be enabled by default before they can be used by most applications. This process leads to a number of problems, the most visible of which is the high effort required and the consequent delay. The lag caused by standardization, implementation by vendors, and widespread deployment is usually measured in years. A complicating factor is that operating system vendors, not third parties, typically must implement upgrades because transport protocols reside in the kernel.

More insidiously, there is a disincentive problem due to "externalities." The benefit of upgrading an individual endpoint depends on the number of other endpoints that have already been upgraded, since the new protocol is not effective unless both have it. This means that early adopters will gain little benefit, with the result that innovation is further discouraged. The only current recourse to this problem is the development of "fully backwards-compatible" upgrades that provide benefit even if only a single end is upgraded. Unfortunately, the pressure to produce such upgrades may result in "quick fixes" that are less effective or robust than the alternatives. For example, both NewReno [20] and SACK [41] protect against multiple packet drops in a window of data. It is widely accepted that SACK is the superior solution [18] and was proposed first, yet NewReno, the fully backwards-compatible scheme, is the most widely deployed solution today. Moreover, this approach cannot be made to work for many proposed upgrades, such as Connection Migration [54]. We are not the first to recognize this general problem. In 1991 the authors of RFC 1263 [42] made many of these arguments, motivated by the many TCP extensions being proposed even then.

Despite these difficulties, upgrades to transport protocols are continually required. TCP has been changing at a steady pace ever since it was designed two decades ago [49] and shows no signs of slowing down; the Internet is very much in flux rather than completed. This is because the trends for growth and change are driven by trends in the underlying technology and set of applications, and so will in all likelihood continue. There is much ongoing research that anticipates the changes in network protocols that will be necessary with further growth and evolution of the Internet: such research includes work on streaming media applications [37, 27] and congestion control of aggregates [8, 6].

To address the continual need for change, we present a system that tackles the compatibility, incentive, and delay problems associated with transport protocol evolution. Our approach is to use untrusted mobile code to allow one host to upgrade, as needed, the transport protocol code used by its peer [46]. This means that installing new functionality at one node produces an immediate benefit, reducing both the disincentive to be an early adopter and the incentive to craft fully backwards-compatible solutions. To reduce the delay in upgrading even an individual node, we allow users or applications to provide their preferred transport protocol implementations to the local node, rather than only permitting operating system vendors to implement new protocols. The common link between these two steps is that transport protocol implementations must be able to function well despite being untrusted by the local node.

We call our system STP, for Self-spreading Transport Protocols. It uses a TCP-like connection setup as a mechanism for triggering the out-of-band deployment of new transport protocols. These protocols can be either classic TCP extensions or any transport protocol that fits between the STP interface in the kernel (Section 3) and the socket interface to applications. STP has the following key characteristics:

- It supports a wide variety of unicast transport protocols because it imposes few constraints on the packet formats used by endpoints.
- It enables an endpoint to protect its integrity and resources without trusting either its peer or the authors of transport protocols.
- It enables the sender to ensure that transport protocols compete reasonably with TCP (an important consideration for new transports [21]) without trusting them.
- It runs transport protocols while incurring only modest computational cost over standard TCP. This means that new transports can realize performance improvements.
- It is compatible with TCP connection setup and applications that use the socket interface. TCP connection setup triggers protocol deployment, but does not delay setup. These aspects provide a bootstrap path for STP deployment.

Our insight is that the domain of unicast transport protocols affords simplifications that make tractable many of the performance and safety problems that must be solved to build a useful mobile code system [46]. Mobile code has long been known as a mechanism to provide a late binding of function to systems [35, 34, 11]. However, it has typically brought its own set of difficulties in terms of security and efficiency to systems as diverse as Java applets in Web content [16] and active networks [62], with the result that few distributed systems in widespread use today depend heavily on mobile code. In our domain, however, TCP connections have stylized interactions with the rest of the operating system that inherently limit buffering, code invocation, and sharing across connections. Each of these restrictions helps to secure the efficient execution of mobile code. We have further leveraged recently developed tools and techniques from networking and programming languages. We use Cyclone [32], a type-safe C-like language, to obtain protection at a reasonable performance cost as well as easy integration with rest of the kernel. We define network safety in terms of TCP-friendliness [48] and check that transports are adhering to it by using the ECN nonce signaling mechanism [17].

The rest of our paper is organized as follows. In the first part, we describe the system that we have built. We begin with the problem we are solving in Section 2, followed by the architecture and design of our solution in Section 3. In Section 4, we describe some key points of the implementation of our system in the BSD kernel. In the second part of the paper, Section 5, we concentrate on an evaluation of the system. We evaluate the effectiveness of our TCP-friendly rate enforcement mechanisms and show that the performance of transport code in our system is competitive with native, in-kernel BSD implementations across the range of typical Internet conditions. In the last part of this paper, we focus on what we have learned and future directions. We discuss key issues based on our experience to date in Section 6, contrast STP with related work in Section 7, and conclude with future directions in Section 8.

2. MOTIVATION

New transport protocols and changes to the dominant transport protocol, TCP, often require upgrades to both connection endpoints before they are effective. This is presently done with independent upgrades to standard protocols at individual endpoints. We have argued this leads to large deployment delays from standardization and implementation, discourages innovation and early adopters who gain little benefit, and encourages the proliferation of "quick fixes" that may sacrifice robustness or efficiency.

To assess the extent to which these factors matter in practice, we surveyed TCP extensions and alternative transports that have been deployed or proposed since congestion control was first introduced in 1988 in TCP Tahoe [29]. We analyzed a total of 27 TCP extensions and transports and classified them into three categories according to which endpoints must be upgraded to gain a benefit, assuming TCP Tahoe as a baseline implementation. The results are shown in Table 1. We found that 16 of the 27 extensions, which are listed in Category 1, require upgrades to both endpoints to be of value. This can lead to adoption times measured in years, as in the case of TCP Selective Acknowledgments (SACK). SACK was standardized by the IETF in 1996 [41] and a study in 2000 reported that 58% of wide-area connections were started by an endpoint capable of SACK but only 5% of the connections were actually able to use SACK [22, 44].

For the five extensions listed in Category 2, upgrades to a single endpoint do provide benefit. However, we observe that all of these extensions have the potential to be either more robust or effective if both endpoints can be upgraded and the new functionality split freely between the sender and receiver. For example, TCP NewReno uses a heuristic interpretation of duplicate acknowledgments to avoid timeouts, but it is widely accepted that TCP SACK (in Category 1) provides better recovery from losses [18]. That is, we argue these extensions are impacted to some extent by the pressure of backwards-compatibility.[1]

Finally, the remaining six extensions in Category 3 inherently require changes to only one endpoint. For example, both the Fast Recovery modification to the sender-side TCP [5] and TCP-Nice [59] are transparent to the receiver. The deployment of such changes is limited primarily by the difficulty of convincing operating system vendors to accept the change and then upgrading and configuring operating system versions, as transport protocols are typically hardwired into the kernel.

The above analysis suggests that the majority of transport extensions and new transport protocols stand to benefit from an upgrade

[1] For the curious, TCP Vegas [14] and TCP Westwood could use receiver timings to more accurately estimate delay measures [47], the retransmission ambiguity could be avoided by coding whether packets were retransmissions, and false sharing due to NAT boxes could be cleanly detected by the Congestion Manager.

Category	Extensions
1. *Require* both endpoints to change	**1. Connection migration**: Migrating live TCP connections [54], **2. SACK**: Selective acks [41], **3. D-SACK**: Duplicate SACK [25], **4. FACK**: Forward acks [40], **5. RFC 1323**: TCP extensions for high-speed networks [31], **6. TCPSAT**: TCP for satellite networks [4], **7. ECN**: Explicit congestion notification [50], **8. ECN nonce**: Detects masking of ECN signals by the receiver or network [17], **9. RR-TCP**: Robustly handles packet reordering [65], **10. WTCP**: TCP for wireless WANs [51], **11. The Eifel algorithm**: Detection of spurious retransmissions [39], **12. T/TCP**: TCP for transactions [13], **13. TFRC**: Equation-based TCP-friendly congestion control [24], **14. DCCP**: New transport protocol with pluggable congestion control [37], **15. SCTP**: Transport protocol support for multi-homing, multiple streams etc., between endpoints [57], **16. RAP**: Rate adaptive TCP-friendly congestion control [52]
2. *Could benefit more* if both endpoints could change	**1. NewReno**: Approximation of SACK from sender side [20] **2. TCP Vegas**: A measurement-based adaptive congestion control [14], **3. TCP Westwood**: Congestion control using end-to-end rate estimation [61], **4. Karn/Partridge algorithm**: Retransmission backoff and avoids spurious RTO estimates due to retransmission ambiguity [36], **5. Congestion manager**: A generic congestion control layer [8]
3. *Require* only one endpoint to change	**1. Header prediction**: Common case optimization on input path [30], **2. Fast recovery**: Faster recovery from losses [55], **3. Syn-cookies**: Protection against SYN-attacks [9], **4. Limited transmit**: Performance enhancement for lossy networks [3], **5. Appropriate byte-counting**: Counting bytes instead of segments for congestion control [2], **6. TCP nice**: TCP for background transfers [59]

Table 1: Classification of TCP extensions, assuming TCP Tahoe as the baseline version.

process that allows both endpoints to be changed at once. Our goal with STP is to provide this "two-ended" upgrade in a practical form. Our approach is to allow developers to freely write transport extensions that require new code at both endpoints and then use mobile code techniques to remotely upgrade both endpoints as the functionality is needed. A further benefit of this model is to speed "one-ended" upgrades by removing operating systems vendors from the path to deployment. We envision the following usage scenarios for STP:

1. A "high performance" TCP is installed along with a Web server, and the corresponding code is pushed to receivers that use the server to provide more rapid downloads.
2. A mobile client installs "TCP connection migration" [54] and ships code to the servers with which it connects to allow itself to move.
3. A network backup application installs "TCP Nice" [59] to perform background transfers. No remote code shipping is needed.

The key challenges in providing this rapid deployment model are to provide flexibility and a platform-independent API without causing security problems or performance degradation. Transport protocol code that comes from sources that are not authoritative — such as the other end of a wide-area connection — should not be vetted according to a trust model. We must ensure that untrusted code cannot compromise the integrity of the host system, consume too many of its resources, or launch remote denial-of-service attacks. Further, standard practice in the networking community is to require that new transport protocols compete fairly with deployed versions of TCP to ensure that they will not undermine the stability of the network [21]. Thus to provide a system that is acceptable in practice we must provide this form of network safety. At the same time, we must allow new transport protocols deployed with STP to be competitive in performance with hard-coded and manually deployed versions. This is because extensions to TCP are often undertaken to improve performance, and STP must introduce little enough overhead that performance benefits can be realized.

3. ARCHITECTURE AND DESIGN

This section outlines the key architectural components, interfaces, and algorithms of the STP framework. We present a general overview of our system, followed by detailed descriptions and explanations of special-case optimizations wherever applicable.

3.1 Overview

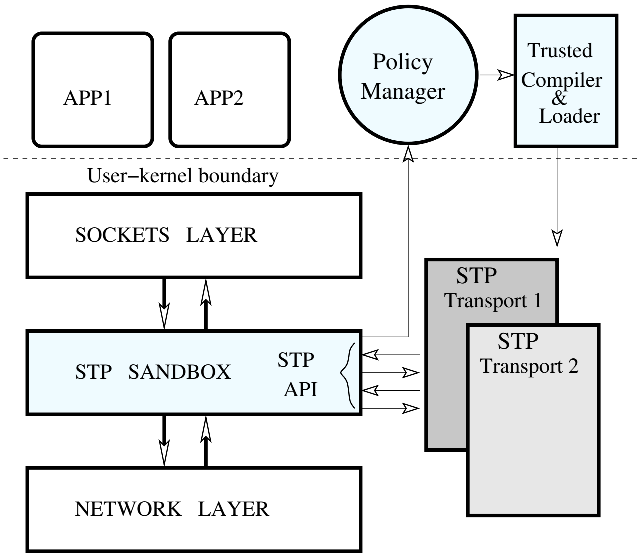

Figure 1: Architecture of the STP framework. The STP sandbox sits below the sockets layer and on top of the network layer. STP provides an API that transport protocols use to get safe access to the sockets layer, the network layer, and certain kernel services. The policy manager uses a trusted tool-chain of compilers, linkers, and loaders to load remote protocol code into the OS kernel.

Figure 1 shows the major architectural components of the STP framework. The central component of the framework, called the *STP sandbox*, provides a restricted and resource safe environment for mobile transport protocols. Safety is achieved through the combination of Cyclone [32], which provides type safety, and careful interposition between the existing *socket* and *network layers* of the operating system. By leveraging a type safe language, the sandbox can ensure that mobile code is unable to reference memory or functions that are not made available to the protocol. These characteristics force the transport protocol to use the *STP API*, a set of functions that provide constrained access to the other layers of the system. For example, a protocol will not be able to open files because the sandbox exports no such function and Cyclone prevents the protocol from linking against random symbols in the kernel. Furthermore, STP supports safe, asynchronous termination of extensions that misbehave, for example, by entering an infinite loop.

Outside of the sandbox there are the user-level programs that make policy decisions, transfer code, and produce trusted kernel modules. The *policy manager* daemon is used to resolve conflicts encountered when making connections to other nodes. The possible outcomes of the policy managers' negotiations are (1) that both sides agree to use a transport that they already have, or (2) that the hosts fail to agree and fall back to using TCP. The former case results in the chosen transport being bound to the endpoints and allowed to proceed. In the latter case, the connection proceeds as a normal TCP connection while the policy managers are free to use out-of-band communication to come to an agreement. Protocols that one side does not possess can then be downloaded, processed by a trusted tool-chain, and made available for future connections. In this manner, code loading activity is amortized across many connections in the expected case that there are few kinds of transports in use, yet many users of the system remain authorized to innovate and possibly develop one of the few widely used transports.

Finally, applications interact with the STP sandbox and transport using the existing sockets interface. Applications wishing to use a particular STP transport set a socket option to the hash that uniquely identifies the protocol. Otherwise, the standard functions, like read and write, behave as one would expect. This allows for an administrator to use a policy manager to map unmodified, legacy applications to selected STP protocols.

The STP API subclasses
1. Protocol management API
stp_load_proto(proto_sw)
stp_unload_proto(proto_handle)
2. Sockets layer API
stp_sowakeup(socket)
stp_sbapppend(socket, seg)
stp_isdisconnecting(socket)
stp_sobind(socket)
3. Connection management API
stp_attach(socket) [callback]
stp_connect(socket, remote-endpoint, state) [callback]
stp_abort(socket) [callback]
stp_accept(socket, remote-endpoint, state) [callback]
4. TCP-friendly network access API
stp_net_send(segment, seqno)
stp_net_resend(segment, new_seqno, old_seqno)
stp_ack_sum(end_seqno, noncesum)
stp_nack(seqno)
stp_net_sendack(segment)
5. Runtime support
stp_gettick()
stp_seg_alloc(proto_handle)
stp_timer_reset(proto_handle, callout)
stp_get_rtentry(proto_handle, dst_ip_addr)

Table 2: Sample functions from the STP API

3.2 STP Components

In the following section, we look at the design rationale for the three main components of the STP framework: the STP sandbox, the policy manager, and the trusted tool-chain.

3.2.1 The STP Sandbox

The STP sandbox is the central part of the STP framework. It implements the STP API and provides a safe and resource-limited runtime to STP transports. Applications, the network layer, and the rest of the kernel communicate with transports through the STP API. Of the 127 functions in the API, 67 are core functions and the other 60 form a library of commonly used kernel-programming support functions. A few sample functions in each major category are shown in Table 2. We give more details of this API as we discuss specific mechanisms and algorithms used by the STP framework.

The API allows transports to register timers, manipulate packet buffers, and interact with the sockets layer in a safe way. Transports are allowed to directly read and write packets from the network, in a manner similar to raw sockets. However, to prevent transports from snooping on packets intended for other protocols or applications, incoming network packets are first classified based on their port numbers and then handed to the appropriate transport code for processing. Similarly, to prevent transports from sending unsolicited packets to arbitrary hosts, the fields of the IP header of outgoing packets are checked. Overall, the STP API is sufficient to implement a range of transport protocols, including conventional TCP.

It is important to prevent untrusted STP transports from affecting the integrity or the availability of the resources of the local host. STP achieves host safety through principles of isolation and resource control similar to those used in language-based operating systems, such as KaffeOS [7]. STP's implementation of these principles is greatly simplified, however, due to the constrained memory allocation and sharing requirements of TCP and similar protocols. Control flow is transferred to untrusted code via three types of system events: socket calls, network packet input, and timers. The CPU and memory usage of all three types of event handlers is tracked and code that exceeds its resource allocation is unloaded from the system. Extensions are never allowed to share memory with other extensions, grab system locks, or disable interrupts. Therefore, asynchronous termination can be safely achieved by closing the sockets that depend on the extension. This tractable notion of termination allows us to use traditional runtime techniques to bound memory usage and an inexpensive timer interrupt to check CPU usage.

Memory safety is achieved by using Cyclone, which has a number of features that make it easier and more efficient to interface with traditional C-based kernels than other safe languages, such as Java or OCaml. First, it has the same calling conventions as C, so C code can directly call into Cyclone code and vice versa. Second, it has the same data layout for basic types and structs, which makes it possible to share data structures between C and Cyclone code without having to copy them. Third, Cyclone supports region-based memory management [26], which can be used to achieve memory safety without the overhead and unpredictability of garbage collection. Fourth, Cyclone has a lightweight runtime that is easy to port into kernels, where runtime library support is generally limited. Finally, Cyclone is syntactically similar to C, which simplifies the translation from existing C code and reduces the likelihood of inadvertent errors. For example, it took us only about one man week to port the FreeBSD implementation of TCP NewReno.

3.2.2 User-Level Policy Manager

STP defers transport admission control and code downloading to a user-level daemon, called the policy manager. The policy manager derives its policies from two sources: those set by a system administrator or user and policies based on the history of a protocol. If a protocol is known to have behaved badly in the past, it is rejected by the policy manager. Once the manager has decided to download a protocol, it is free to select the form (binary or source) and origin of the code. After the code has been received, the policy manager is responsible for determining which restrictions to enforce on the protocol. For example, a transport that is distributed in binary form and signed by a trusted OS vendor could be loaded and executed without any restrictions.

In our current design, a simple policy for foreign code is used: untrusted remote code is accepted unless an application explicitly disables it. This simple policy works in practice because an untrusted transport cannot harm the local host, other hosts on the network, or the network as a whole. At worst, the transport may underperform, in which case the user may flag it as buggy to prevent it from running again. In the longer term, the design of good policies is likely to be an important consideration in controlling the deployment and usage of upgrades. However, we require experience to define those policies, and have deferred the issue in the belief that good policies can evolve as the system matures.

3.2.3 *Trusted Tool-Chain*

Untrusted protocol code is transferred in Cyclone source code form. However, merely writing code in a type-safe language does not make it safe for inclusion in an OS kernel. For example, unrestricted Cyclone code can access arbitrary C code and data, incur stack overruns, or cause a division-by-zero error. STP uses a trusted compiler that prevents transport code from including or accessing arbitrary C code and inserts runtime checks to prevent the code from crashing. The generated protocol module is then loaded into the kernel by a trusted loader.

3.3 STP Protocols and Algorithms

3.3.1 *Transport Negotiation and Loading*

Two important goals of the STP signaling and code loading design are to maintain backwards compatibility with TCP and to allow transports to use new header formats. Although several other designs are possible, in our current design STP achieves these goals by distinguishing between two classes of transports: those that use a TCP-compliant header (XTCP) and those that do not (NTCP). For the NTCP class, STP requires that the transport headers carry the source and destination port numbers in the first four bytes (as in TCP) and use the following byte to indicate whether a packet is for the STP layer or for the transport. The fixed location of port numbers is used for secure demultiplexing of packets to the correct transport. The extra byte required in NTCP headers makes our connection setup protocol resilient to the loss of ACK packets, as described later in this section. Other than these, STP places no restrictions on the format of transport protocol headers. In the following, we discuss the details of our code loading protocol for the XTCP class of transports. It works similarly for the NTCP class but uses a modified header format.

STP accomplishes remote distribution of code with minimal overhead, and full backwards compatibility with TCP, by interposing on normal TCP connection setup. When an application issues a `connect` system call, the STP layer starts a TCP-like three-way handshake with the remote end. It piggybacks transport negotiation options on the connection establishment packets, as discussed below. If both endpoints agree on an already loaded STP transport, the STP layer binds that protocol to the connection and passes the connection state to it. The connection then proceeds normally using the negotiated protocol. If a connection cannot be immediately established using the desired protocol, it falls back to the default TCP implementations on both nodes.

The STP option	Semantic meaning
STP-USE, <proto>	Use a particular STP protocol
STP-OK, <proto>	Agreed to use a particular protocol
STP-SENDME, <proto>	Send a particular protocol using the policy manager
STP-NO	Not allowed to use protocol requested in previous STP-USE

Table 3: STP connection setup options.

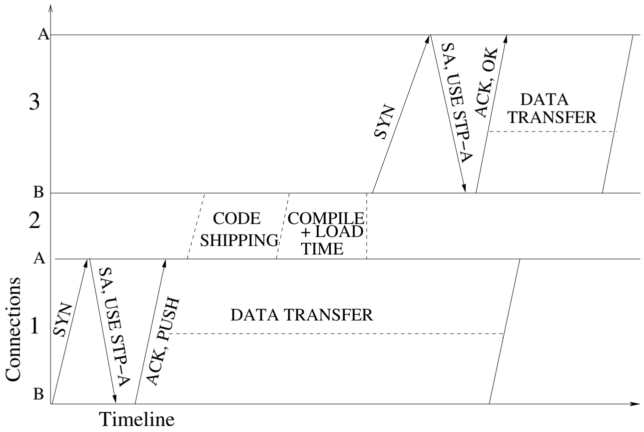

Figure 2: An example scenario. In connection 1, server A requests use of high-performance transport protocol (STP-A), causing client B to ask for the code. At application level in connection 2, server A sends it to client B. In a later, unrelated connection 3, A and B use STP-A.

In the following discussion, assume that initiating host A and listening host B want to open a connection with each other. Node A prefers protocol STP-A while node B prefers protocol STP-B. Table 3 summarizes the STP protocol negotiation options and Figure 2 illustrates a transport negotiation scenario.

At node A, STP prepares a TCP SYN connection setup packet in response to the `connect` request from the sockets layer. It attaches a special STP-USE option containing the code hash for STP-A and sends it to B. The STP sandbox at node B attempts to map it to a loaded protocol. Four different scenarios can take place on B:

1. STP-A is allowed and the protocol is already loaded.
2. STP-A is allowed but the protocol is not yet available for use.
3. STP-A is not allowed.
4. Node B wants to use STP-B instead.

In each scenario, B sends a TCP SYN-ACK packet to A but it attaches a different STP option from Table 3 in response to A's request. B creates compressed state for this connection and stores it in a cache. (Caching responses to SYN packets and thereby deferring socket creation is a standard mechanism used by most operating systems to counter SYN-flood attacks.) In scenario 1, STP sends the STP-OK option. From then on the connection proceeds normally using STP-A. In scenario 2, STP attaches the STP-SENDME option, requesting node A to push the code for STP-A to node B. In scenario 3, STP attaches the STP-NO option, indicating a refusal to use STP-A. In scenario 4, STP attaches the STP-USE option containing the hash of protocol STP-B, requesting node A to use STP-B.

On receipt of a SYN-ACK packet, node A decides which protocol to use based on the STP option received and sends back an appropriate STP-OK option in the ACK packet. In scenario 1, STP-A is used. In scenario 2, it informs the policy manager to send the code for STP-A to B and decides to use the default protocol for this connection. In scenario 3, the default protocol is used. In scenario 4, it asks the policy manager, and if STP-B is allowed and already loaded, it is used, else the default protocol is used. The ACK packet containing the selected protocol information completes the three-way handshake. The STP layer then attaches the negotiated transport to the connection and passes the connection state to it. The transport may then exchange more connection options with its peer before being ready for data transfer. For the XTCP class of extensions, this step may be unnecessary because the STP layer has already negotiated the standard TCP options, such as window size, window scale, initial sequence numbers, and so on.

One issue with the above connection setup protocol is the loss of the final ACK packet from A to B. The acknowledgment packets are not assigned sequence numbers and their loss is not tracked. Therefore, after sending the final ACK packet, node A assumes that the negotiation is complete and it attaches the negotiated transport to the connection. The negotiated transport can then start sending data using its own packet format. For the XTCP class of transports, the STP layer at node B will read the TCP header and the cumulative acknowledgment contained therein. It can then complete the three-way handshake and start the negotiated protocol. For the NTCP class of transports, the STP layer uses the extra byte to infer a response to the SYN-ACK packet. If the STP layer receives a packet with this byte set to a non-zero value, it starts the negotiated protocol and passes the packet to it. If the byte is zero, then STP assumes that the other end has agreed to use the default protocol.

3.3.2 TCP-Friendly Rate Limiting

Network resource control is cooperatively implemented by endpoints in the current Internet. Endpoints interpret packet loss as signaling congestion and reduce their sending rate accordingly so that flows compete for a fair share of the resources instead of overloading the link. New transport protocols, deployed with STP or otherwise, should limit themselves similarly [21]. However, in our context, we cannot trust the transport and so STP must enforce an upper limit itself. Currently, this upper limit is the same as TCP's, whose behavior is modeled by the TCP rate equation [43], shown as Equation 1.

$$T = \frac{s}{R * \sqrt{2 * \frac{p}{3}} + (t\_RTO * 3 * \sqrt{3 * \frac{p}{8}} * p * (1 + 32 * p^2))} \quad (1)$$

This equation gives an upper bound on TCP's sending rate **T** in bytes/sec, in terms of mean packet size **s**, loss event rate **p**, round-trip time **R**, and retransmission timeout t_RTO. (Note that for determining **p**, all packets lost in a single RTT are encompassed by a single loss event.) Previous research efforts [64, 43, 27] have demonstrated, in both simulation and real-world implementation, that the sending rate derived from this equation competes fairly with normal TCP. Therefore, this equation forms a promising starting point, though it is conceivable that it could be replaced as we gain more experience with the system.

A key difficulty is that the equation depends on parameters, such as packet loss rate, which must be derived without trusting the local transport protocol or the remote host. To accomplish this, the transport supplies STP with sequence numbers and acknowledgments that are checked using the recently proposed ECN nonce mechanism [17, 50]. Round-trip times come from the process of sending packets and receiving the corresponding acknowledgments. Packet loss data is either supplied by the protocol (and checked by STP) or inferred by STP when transmissions are not acknowledged after a certain amount of time. The verification process relies on a random one-bit nonce that must be given by the sender when informing the STP layer of an acknowledgment, as is described next. The checking mechanism is probabilistic, but has been shown to work reasonably for limiting transport performance even against an adversary [17]. Note that our checking mechanism does not require the deployment of ECN at routers. ECN nonces are primarily used as proof of acknowledgment; congestion signaling, if available, is an added benefit to the transports.

The ECN nonce mechanism works as follows. A transport protocol assigns a unique sequence number to each outgoing data packet before sending it using the `stp_netsend` function in Table 2. The STP runtime then places the random one-bit nonce, unknown to the sender, in the IP header of the outgoing packet in a standard location. STP also maintains a history of unacknowledged sequence numbers along with their nonces. These nonces will be lost if the packet itself is lost in the network or if an ECN-capable router signals congestion by marking the packet. Otherwise, the nonces will reach the receiver who is then expected to echo them back when acknowledging packets. Rather than echoing nonces for individual packets, which would not be resilient to packet loss, the receiver sends the running sum of nonces along with an acknowledgment. Acknowledgments and their nonce sums are then presented to the sender's STP layer as proof that the packets reached their destination without being dropped; packet losses are inferred when the received nonce sum does not match what was sent. If a loss is detected by the sender, a proactive transport will report a negative-acknowledgment to the STP layer and resend the packet using `stp_netresend`. Timely reporting of losses to the STP layer results in a rate limit that closely matches the current network conditions. Otherwise, STP will infer the loss after a long delay, thereby inducing a lag in the limit predicted by the equation and the actual network state. Finally, the round-trip time, packet loss, and other data derived during this process are fed back through the equation to determine when the next transmission will be allowed.

3.4 Flexibility and Limitations of STP

Clearly, the STP framework cannot support all types of existing and future transport protocols. The following are the key flexibilities and constraints of STP:

1. **Packet Format.** STP demultiplexes incoming network packets to the correct transport protocol by first classifying them based on their port numbers. This implies that STP needs to understand the transport protocol header to look up the port numbers. This is not a problem in practice because most known transport protocols (including TCP, UDP, DCCP, and SCTP) carry the source and destination port numbers in the first four bytes of the transport protocol header.

   STP imposes no further constraints on the format of the packets or headers. Transports are free to carry the sequence numbers, acknowledgments, and the rest of the header in any format they like. The rate control mechanism is independent of the sequence number formats that are carried in packets. The freedom to exchange new information in new formats (in packet headers) and to ship protocol code to process that information are two features that arbitrary new protocols require.

2. **Packet Loss Signals Congestion.** STP checks that transports compete reasonably with TCP. This, in turn, means that loss

must be treated as equivalent to congestion. Transport protocols for high loss-rate wireless networks, such as Wireless Transport Control Protocol (WTCP) [51], do not treat losses as congestion and hence cannot be accommodated in our current design.

The above limitation could be overcome by trusting the receiver's end and carrying an extra STP header in packets.[2] The receiver side STP layer could distinguish between congestion marks and losses, and report the correct congestion event rate to the sender. A similar receiver-side algorithm is also used by TFRC [27] to calculate the parameters of Equation 1. However, in our context, this design has security implications and it imposes extra network overhead—the STP header will carry redundant information already present in most transport headers.

3. **TCP-Friendliness.** STP uses the TCP response equation [24] to limit the sending rate of untrusted transports. A potential concern is whether this equation will be able to accommodate newer transport protocols. For example, HighSpeed TCP (HSTCP) [23] and FAST TCP [33] are recent transport protocol proposals designed to be more aggressive than TCP-friendly transports under very high speed (e.g., 10Gbps) and low loss rate (less than 10^{-3}) network environments. For example, HSTCP switches to a high speed TCP response function when the congestion window size exceeds a certain threshold. Because STP already measures the required parameters, it appears that it can easily adopt such a response function, and thereby support these high speed transports.

4. IMPLEMENTATION

We have implemented a prototype of the STP framework in the FreeBSD kernel version 4.7. Most of the framework, including the core STP API, is implemented in C. Cyclone wrappers export the API to untrusted transports. The current implementation consists of approximately 6000 lines of C code and 1000 lines of Cyclone code.

Transport extensions implement a set of functions similar to those implemented by native BSD transports like TCP and UDP. New protocols have control over input and output processing, and can register software timers. However, extensions are not trusted to implement proper socket creation or teardown, nor are they trusted with routing decisions or with demultiplexing incoming packets to the correct protocol.

4.1 TCP-Friendly Rate Limiting

STP implements network rate control as defined by Equation 1. Table 4 describes the state transitions for a new socket. When a connection starts up, no data has been exchanged, and the system has no estimate for the round-trip time. During this INIT stage the protocol is limited to sending 4 packets per second, which corresponds to the initial congestion window on BSD. Once acknowledgments begin to arrive, the protocol enters the SLOWSTART phase and can send twice as many packets each RTT. Once the first loss is encountered, the socket enters STEADYSTATE, and its throughput is governed by the throughput equation. Figure 3 outlines this process in pseudocode.

Once in STEADYSTATE state, the connection is subjected to three types of checks. The first check is performed at each packet send

Socket State	Semantic Meaning
INIT	Socket has started sending packets but there haven't been enough acknowledgments to measure RTT.
SLOWSTART	Socket hasn't taken a loss; is allowed to double sending rate each RTT.
STEADYSTATE	Socket has taken a loss; sending rate governed by throughput equation.

Table 4: STP socket states.

to ensure that a protocol does not increase its sending rate by more than a constant factor (currently set to 1.3, based on our experience) times the rate calculated by Equation 1. The rate calculation is shown in Figure 3. A value of greater than one for the constant factor ensures that the rate limit is not too restrictive at the granularity of each packet. A second check that ensures that our rate enforcement is not too liberal at a coarse granularity is performed every eight RTT intervals. This check forces a protocol's average sending rate to comply with the rate control equation. If a protocol's average sending rate consistently violates the rate control limit, the protocol is terminated. A third check is used to ensure that congestion information is retired for idle connections. If a protocol has been idle for four RTTs, its sending rate is cut in half, eventually forcing the connection back into slow-start.

```
loss = loss_rate();
if (loss == 0) {
  /* The connection is in slow-start */
  rate = 2 * prev_acks;
} else {
  /* The connection is in steady-state */
  rate = 1.3 * equation1(loss, rtt, rto);
}
rate = max(rate, SLOWSTART_WINDOW);
```

Figure 3: Calculation of allowed send rate. STP performs this calculation once every RTT. `prev_acks` is the number of packets acknowledged in the previous RTT. Function `loss_rate` calculates loss event rate, and function `equation1` applies Equation 1 to its parameters.

4.2 Host Safety

STP sandboxes untrusted protocols to prevent misuse of system resources. Sandboxing relies on the ability to terminate a protocol to regain control. For the domain of transport protocols, termination is simplified by the fact that transports do not share data with other transports. Therefore, terminating a transport does not result in any missing references and only affects the connections that are using the transport. In STP, normal termination of a protocol is performed by calling a protocol-specific cleanup routine for all its connections. The protocol is allowed up to two maximum segment lifetimes (MSL), typically set to two seconds, to cleanly tear down a connection, after which all resources held by a protocol are reclaimed and it is unloaded.

4.2.1 Memory Control

STP limits the use of memory resources by performing runtime checks on every allocation made by untrusted code. Each call to the memory allocator routines includes a reference to the protocol that is charged for the allocation. If a protocol's memory allocation

[2] The extra STP header is required so that the STP layer does not need to rely on transports and understand their header formats for piggybacking its data on packets.

exceeds the allowed limit, the protocol is denied any more memory and is flagged as violating the memory limit. If the number of such unsuccessful requests exceeds a limit, the protocol is stopped and unloaded.

To prevent transport code from making illegal memory accesses, all memory belongs to one of two top-level memory regions [26]: a kernel region and a per-transport region. The transport code is not allowed to store references to kernel memory in per-transport memory. Therefore, a transport cannot create dangling pointers and use them to read arbitrary kernel memory or trigger faults.

We plan to adapt the next version of the OKE Cyclone compiler [12] to implement compiler support for these regions. We are currently using version 0.4 of the Cyclone compiler, which is not compatible with version 0.1.2 on which the current OKE Cyclone compiler is based. The OKE compiler also addresses two limitations of the current Cyclone compiler, preventing stack overflow and numeric traps by performing static analysis and inserting minimal runtime checks. We believe that these changes will only slightly affect the performance numbers presented in this paper.

4.2.2 CPU Controls

STP implements CPU controls by using a modified timer interrupt handler. At entry points into the untrusted transport, a flag is set and the current timestamp is recorded. The flag is unset at exit from the protocol code and the difference between the current time and the timestamp is charged to the transport. If a transport's CPU usage exceeds the allowed limit, an exception is raised, and a trusted handler invokes a cleanup routine to stop and unload the protocol. In the event of such a CPU violation, the transport is not allowed to cleanly shutdown all its connections. In the future, we may explore techniques to statically limit the runtime of transport code, similar to our hybrid resource control work [45].

5. EVALUATION

In this section, we present an evaluation of STP. First, we describe the transports we developed and argue that STP is expressive enough to support a wide range of extensions. Second, we demonstrate that STP transports compete reasonably with TCP and do not overload the network. Third, we show that protocol code is small enough and loads quickly enough that our decision to ship complete protocols is practical. Next, we show that it imposes low computational overhead. Finally, we show that the performance of STP extensions is competitive with native transport protocols over a range of network conditions.

Experiments were conducted on Netbed's Emulab [63] cluster network testbed facility. For all except gigabit experiments, we used 850MHz Intel Pentium IIIs with 512 MB of SDRAM and five Intel EtherExpress Pro/100+ PCI Ethernet cards. For the tests involving gigabit network interfaces, in order to avoid PCI bottlenecks, we used 1.8 GHz Intel Xeons with Intel Pro/1000 cards on a 64-bit, 66 MHz PCI bus.

5.1 Transports

To evaluate our system, we surveyed a large number of TCP extensions, including several in detail, ported two transport protocols to STP, and developed a third. The three that we fully implemented are TCP NewReno, TCP SACK, and UDP Flood. In both its C and Cyclone forms, UDP Flood is about 1000 lines of code, while each TCP version is approximately 10000 lines. These counts give some indication of the required implementation effort. Working with Cyclone was not difficult: it took only six person-days, split between two people, to convert the first STP version of TCP to Cyclone. One of the programmers had never worked with Cyclone or TCP before, and the six days included at least one day's effort enhancing and fixing the underlying STP layer. We find this effort surprisingly modest, and it indicates that this approach is practical from the software engineering point of view.

5.1.1 UDP Flood

UDP Flood is a UDP-like unreliable transport that sends data as fast as STP will allow. UDP Flood differs from UDP in that it sends back acknowledgments for received data. Without this change, the policer would assume every packet is lost, and UDP would be unable to send any traffic. Because UDP Flood has no congestion control of its own, it shows how STP can be used to construct TCP-friendly transports.

5.1.2 TCP NewReno

We first ported the standard TCP implementation in FreeBSD 4.7, TCP NewReno [20], to the C and Cyclone versions of the STP API. NewReno is widely deployed in the Internet; in their 2001 survey [44] Padhye et al. found it to be the most prevalent variant of TCP congestion control. NewReno differs from the classic TCP congestion control algorithm in that it uses partial acknowledgments to avoid retransmit timeouts. The only STP-specific change to the standard behavior was to disable "delayed ACKs" so that round-trip times could be inferred correctly. Supporting "delayed ACKs" would require that the STP layer on the sender side trust timestamp values in the acknowledgment that indicate the length of the delay. This conflicts with one main design goal of STP: the sender side should not be required to trust the receiver side. To level the field in the performance measurements, "delayed ACKs" were also disabled in the standard version. Because of its superior performance, NewReno represents a good test of the flexibility of the STP rate control mechanism.

5.1.3 TCP Selective Acknowledgments

We modified the FreeBSD implementation of TCP NewReno to support TCP selective acknowledgments (SACK) [41] and then ported it to the C and Cyclone versions of the STP API. SACK is a variant of TCP that improves performance when multiple packets from a single window of data are lost. In classic TCP, the sender can only learn about a single lost packet per round-trip time. This limits the ability of the sender to quickly recover from multiple losses within a single round-trip. In TCP SACK, the receiver informs the sender of which segments were received so that the sender can make intelligent decisions about retransmissions. The implementation of SACK highlights the flexibility provided by the STP framework in choosing packet formats and options: the sender and receiver can use any header format they want as long as they carry the nonces required by STP.

5.2 Expressiveness

To evaluate the expressiveness of the STP API, we examined STP's support for the new transport protocols and proposed TCP extensions listed in categories 1 and 2 in Table 1. We analyzed whether each protocol could perform effectively if written to the STP API. The results of our analysis are presented in Table 5. Most of the extensions, 18 out of 21, can be effectively supported.

All the extensions can be easily programmed using the STP API. However, as noted in Section 3, STP does not support the semantics of non-TCP-friendly protocols. Therefore, current STP does not support TCP Westwood [61] and TCP for wireless WANs (WTCP) [51]. DCCP is a new unreliable transport protocol that lets applications choose from a set of congestion control algo-

Extension	Support
Connection migration [54]	✔
SACK [41]	✔
D-SACK [25]	✔
FACK [40]	✔
RFC 1323 [31]	✔
TCPSAT [4]	✔
ECN [50]	✔
ECN nonce [17]	✔
RR-TCP [65]	✔
WTCP [51]	**–**
The Eifel algorithm [39]	✔
T/TCP [13]	✔
TFRC [24]	✔
DCCP [37]	✔ / **–**
SCTP [57]	✔
RAP [52]	✔
NewReno [20]	✔
TCP Vegas [14]	✔
TCP Westwood [61]	**–**
Karn/Partridge algorithm [36]	✔
Congestion manager [8]	✔

Table 5: STP support chart for TCP extensions in categories 1 and 2 from Table 1. WTCP and TCP Westwood are not TCP-friendly; connection migration and SCTP require a special API to securely change remote endpoint addresses; DCCP may potentially run a congestion control profile more aggressive than standard TCP; T/TCP, TFRC, DCCP, and SCTP require changes to socket-based applications.

rithms. The STP API will support only TCP-friendly instances of DCCP.

5.3 Network Safety

The role of the STP network policer is to enforce that all transports obey TCP-friendly congestion control. To evaluate the policer, we used an application that sends as fast as possible using the UDP Flood transport. Because UDP Flood uses no congestion control, its throughput represents a worse-case upper bound for bursty traffic. We used Dummynet [53] to emulate wide-area links across a range of conditions.

Figure 4 shows the average throughput of UDP Flood flows across a range of RTTs and packet drop rates. The link capacity was set to 10 Mbps, and each data point represents the average of about 180 one second samples. We compared each UDP Flood flow to a standard NewReno TCP flow running in identical conditions. Each flow was run by itself. The results show that STP restricts UDP Flood's bandwidth consumption to a level modestly above TCP. The throughput difference averages 18%, with a maximum value of 36%. The UDP results exhibited an average coefficient of variation of 13% at .2% loss and 20% at 1% loss. UDP's variation did not depend on RTT, and was smaller than the corresponding TCP variation.

Next, we evaluated the STP rate-limiter using real background traffic in place of artificial drops. We configured three normal FreeBSD (not STP) TCP flows to compete for a WAN-like shared link of 10 Mbps capacity and 50 ms RTT. Figure 5 shows the throughput attained by a fourth flow, which was either a UDP Flood flow, an STP-based TCP flow, or a baseline TCP flow using native FreeBSD. Each of these experiments was conducted separately.

The performance of the TCP flow in STP is indistinguishable

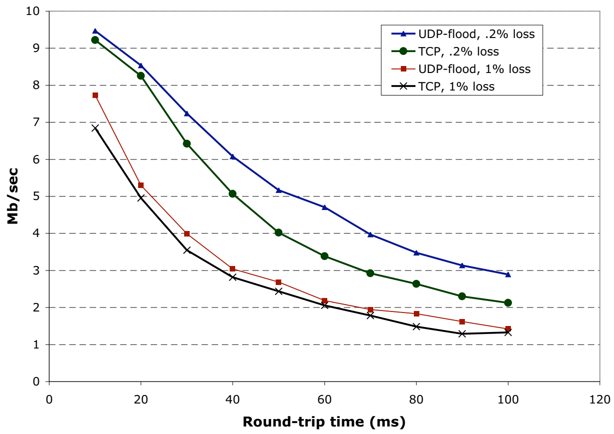

Figure 4: UDP Flood versus NewReno TCP: the STP policer restricts a UDP flow's throughput to within 36% of TCP's on a 10 Mbps link.

Element	Source Bytes	Binary Bytes C	Binary Bytes Cyclone	Source Lines C	Source Lines Cyclone
Kernel		964,780			
TCP	86,670	22,959	30,680	10,388	10,396
TCP SACK	94,829	23,617	33,317	11,200	11,507
UDP Flood	10,215	3,629	6,659	1,066	1,054

Table 6: Mobile code size. This table shows the gzip-compressed size of the Cyclone sources, FreeBSD kernel, and kernel modules for the three transports in both languages. Also shown are the source line counts for the C and Cyclone versions of the transports.

from the native FreeBSD flow. This indicates that our policer does not unduly punish TCP-friendly flows. The UDP Flood application is limited to approximately its fair bandwidth share over the lifetime of the trial. However, UDP exhibits substantially more short-term variation in throughput. The coefficient of variation for UDP is 60%, compared to only 10% for the two TCP trials.

We believe this erratic performance is due to UDP's interaction with TCP dynamics. Small timing effects can have a large impact on TCP performance—for example, pacing has been shown to noticeably degrade TCP throughput [1]. More mature transport protocols such as TFRC [27] have paid considerable attention to the averaging intervals for loss-rate information and the inter-packet spacings to minimize such rate variations. Policing with improved short-term stability is an important area of future work. Another area to examine is the apparent discrepancy between UDP Flood's average throughput in the two sets of experiments. This difference may be related to Dummynet's uniform distribution of loss.

5.4 Code Shipping and Loading

Our current approach treats entire transport protocols as the minimal unit of extensibility, and therefore mobility. To evaluate the practicality of shipping such seemingly large units of code across networks, we measured the sizes of our Cyclone-based STP transport implementations and gathered timings. Source code size is relevant for systems running a trusted compiler; object code size is relevant for systems that accept code signed by a trusted provider. The object code retains external symbol information, which is required to load it dynamically.

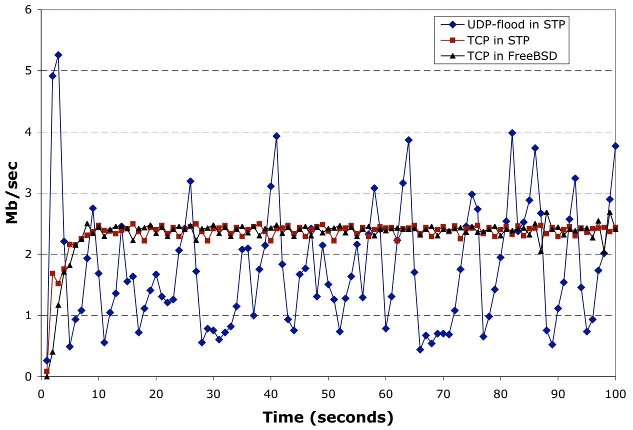

Figure 5: Throughput with three background TCP flows: The throughput of a fourth flow competing against three background TCP flows. All experiments were conducted separately. Each flow ran on a 50 ms RTT link with a 10 Mbps bottleneck.

Table 6 shows that the gzip-compressed source for full TCP transports are 87–95KB, while the compressed object files are 23–33KB. In today's relatively high speed and pervasive networks, and the background manner in which we do code distribution, these sizes clearly pose no deployment problem. Indeed, the binaries are small enough to be practical to deploy over wireless networks on small devices. For comparison, the corresponding compressed FreeBSD kernel binary is almost 1 MB in size, meaning that our largest unit of mobile code, the SACK transport, is less than 3.5% of the entire kernel. This is more evidence that protocol-level granularity is reasonable. Numerous protocols can be installed and still represent only a small portion of the overall system.

We measured how fast our (entirely unoptimized) prototype system performs automatic shipping, compiling, and loading of real protocols. On an 850 MHz Pentium III, over a 2 Mbps link with 100 ms RTT and 0.5% packet loss rate, the end-to-end processing of an entire NewReno STP implementation in Cyclone source form takes less than 12 seconds. Upon triggering by an STP-SENDME option, shipping takes less than a second, compiling takes about 10 seconds, while loading takes less than 0.5 seconds.

5.5 Base Overhead

Each socket requires its own STP protocol control block, which contains 196 bytes of metadata and a scoreboard data structure. The scoreboard stores 16 bytes of rate control information for each unacknowledged packet—a relatively modest overhead, given that full-sized data segments exceed 1400 bytes. Since our current implementation statically allocates a 2048-entry scoreboard, our per-socket overhead is just over 32 KB, and 100 active sockets would require less than 3.3 MB of memory.

To assess CPU overhead, we measured the fraction of kernel CPU consumed by a full-speed flow on a 1000 Mbps link, as shown in Table 7. We evaluated three versions of TCP: native BSD NewReno, STP NewReno in C, and STP NewReno in Cyclone. All versions achieve roughly equal throughput, 895 Mbps, using 8258 byte Ethernet frames. The Cyclone version requires 24% more CPU time for the sender and 54% more time for the receiver (the last column of ratios in the table).

TCP Version	FreeBSD	STP–C (Ratio to BSD)	STP–Cyclone (Ratio to C, Ratio to BSD)
Gigabit jumbo frames on 1.8 Ghz Xeon			
Sender CPU	58.7%	59.2% (1.01)	73.0% (1.23, 1.24)
Receiver CPU	47.5%	61.2% (1.29)	73.0% (1.19, 1.54)
100 Mbps on 850 Mhz P-III			
Sender CPU	24.7%	29.0% (1.17)	35.6% (1.23, 1.44)
Receiver CPU	20.1%	21.7% (1.08)	26.5% (1.22, 1.32)

Table 7: Kernel CPU usage and ratio to the CPU usage of faster implementations: for a single flow between a sender and receiver, for three implementations of TCP. The median of five trials was selected, with the times only varying by +/- 1%. Times were measured using FreeBSD's internal global counters over a 20 second period. The user-mode CPU usage was negligible in all cases.

TCP Version	FreeBSD cycles	STP–C cycles (Ratio to BSD)	STP–Cyclone cycles (Ratio to C, Ratio to BSD)
Sender Input	10665	13320 (1.25)	20016 (1.50, 1.88)
Sender Output	3800	4646 (1.22)	6774 (1.46, 1.78)
Rcvr Input	10108	10224 (1.01)	16109 (1.58, 1.59)
Rcvr Output	2096	2103 (1.00)	3907 (1.86, 1.86)

Table 8: Cycles per I/O function and ratios to faster implementations: for a single 100 Mbps flow between a sender and receiver for three implementations of NewReno TCP, on an 850 Mhz P-III. The median of five trials was selected, with the cycle counts only varying by +/- 1%. Times were measured using the Pentium cycle counter over a 20 second period. In all cases, the user-mode CPU usage was negligible.

We performed a similar set of CPU measurements over a 100 Mbps link, shown in the lower half of Table 7. As for gigabit links, all three flows were able to saturate the link. We found that the Cyclone sender and receiver take 44% and 32% more CPU than the FreeBSD TCP implementation. The majority of the overhead is due to Cyclone processing: 27% for the sender and 24% for the receiver. Table 8 shows further detail on CPU use, broken down by input and output functions on both sides of the connection. We also measured the overhead of the rate-limiter on the sender side and found it to be modest: only 492 and 650 cycles in the output and input functions, respectively.

As we see from the numbers, Cyclone costs are significant. A major portion of the Cyclone costs stem from our functions that marshal data between the kernel and Cyclone code. Much of this overhead could be eliminated by inlining C functions in Cyclone code, but the version of the Cyclone compiler that we used (0.4) could not successfully inline them. By temporarily turning off Cyclone's range and null pointer checks, we found that they cost only 5% of the total execution time. Overall, Cyclone is a relatively young compiler, and STP should be able to ride future performance improvements. More detailed examination of the sources of overhead remains as future work.

5.6 Overall Performance

In this section, we evaluate how the overhead of STP affects the performance of transports under a range of typical wide-area (WAN) and local-area (LAN) network conditions. We compare the performance of two implementations of TCP SACK and TCP NewReno (TCP): an STP implementation and a hard-wired implementation in the FreeBSD 4.7 kernel.

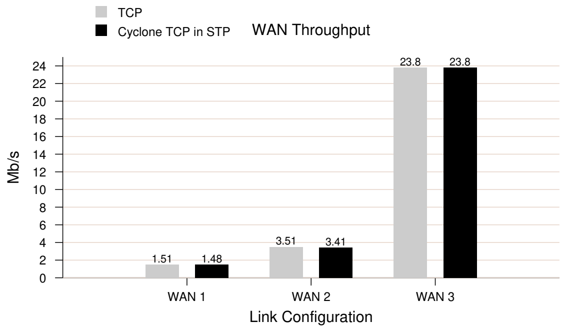

Figure 6: WAN performance of NewReno vs. NewReno in STP

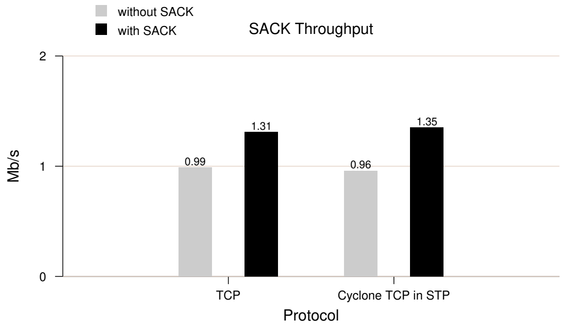

Figure 7: WAN performance of SACK implementations

We measured these transports under three different wide-area network conditions: a 2 Mbps link with 100 ms round-trip latency and 0.5% packet loss rate (WAN 1), a 5 Mbps link with 50 ms RTT and 0.5% packet loss rate (WAN 2), and a 25 Mbps link with 50 ms RTT and 0% loss rate (WAN 3). The results shown in Figure 6 are the median of five trial runs. In all three configurations we found the throughput of the STP implementation to be roughly the same as the hard-wired implementation. These results indicate that, at least on desktop machines with a limited number of simultaneous connections, the moderate overheads of the STP rate limiter and Cyclone will not harm TCP's throughput in the wide-area Internet.

We performed another set of wide-area experiments to verify that the performance improvements due to new transport protocols, in this case SACK, can be realized with STP. We created an environment in which the performance improvements due to SACK are visible: a 2 Mbps link, 100 ms round-trip time, and 0.5% packet loss rate with bursts—the router probabilistically drops packets in bursts of two to four. Figure 7 shows that the SACK in STP retains its performance benefits, doing much better than NewReno in STP, even slightly exceeding FreeBSD SACK's performance.

Since STP's computational overhead should be most visible in high-speed network environments, we measured three different implementations of NewReno on a 1000 Mbps LAN. We performed these experiments with both large Ethernet frames (8258 byte Jumbo Frames) and regular 1514 byte frames. The results are shown in Figure 8. With small frames even FreeBSD's TCP cannot saturate the link. The overhead of the STP API, the bulk of which is the rate limiter, reduces TCP throughput by an additional 12%, while Cyclone drops it 8% more, sending data 20% more slowly than TCP. The use of Jumbo Frames reduces the overall load, letting all three TCP implementations achieve essentially identical throughput, about 895 Mbps. This result demonstrates that desktop-class machines can achieve high-speed data transfers in the STP environment.

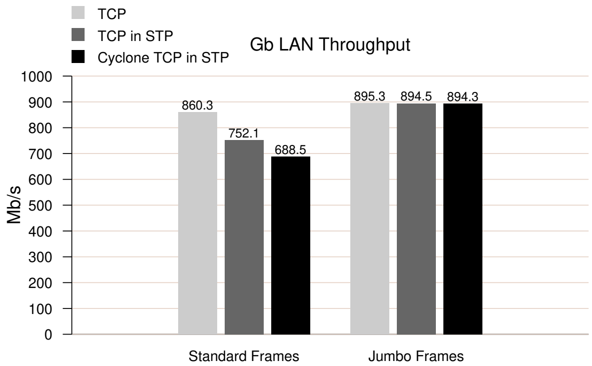

Figure 8: Gigabit LAN performance of three NewReno implementations

6. DISCUSSION

Our initial experiences developing STP and its applications have helped us to gain an understanding of the strengths of our design and the difficult issues that remain. We briefly discuss several of these issues here.

Expressiveness of the STP API: It is important that new, unanticipated transport protocols be compatible with the STP API or our system will be of limited value. Only time will tell the extent to which this is true. However, we have been pleasantly surprised by how many of the extensions we considered could be coded to our initial API. We believe the reason for this, and a key strength of the design, is that STP minimally constrains the form of information passed between the communicating endpoints. Many TCP extensions boil down to the need to pass different information between sender and receiver, e.g., about received packets, or to use the available information differently, e.g., TCP Vegas. This suggests that STP has the potential to support a large class of transport protocols. There are also some obvious steps that we have not yet taken, such as a UDP-style socket interface to applications that would support unreliable transports. Multicast transports may also be possible, though they seem a large departure from our model unless positive acknowledgments are used.

Extension Granularity: One issue that remains to be explored is the granularity of extensions. Our current design provides extensibility at the granularity of an entire transport implementation. Although we originally took this route for expedience, in retrospect it may be the right approach for the much longer term. This coarse-grained design provides complete flexibility and is practical in most domains, as the code sizes demonstrate. It clearly has the additional virtues of simplicity, avoidance of feature interaction, early deployment, and low barriers to adoption. It does have the downside of increased disk and memory use, especially over time, as implementations accumulate. We have considered a number of designs for a more modular implementation, that would allow incremental and composable extensions to (at least) TCP. There is much related work such a design could leverage, e.g., Prolac [38] and the Fox project [10]. The tradeoffs between these two approaches are similar to the tradeoffs in shared library (DLL) deployment. With the inevitable march of Moore's law, in recent years the cost that DLLs bring in complexity has grown to exceed their advantages in disk and memory savings. This has led vendors such as Microsoft gradually to move away from them. We believe that such may also be true in our STP domain.

Network Safety: We leveraged recent work on TCP-friendliness as the basis of our network safety. This has worked well, particularly given the synergy with the ECN Nonce: an STP sender can check that a transport protocol is not sending faster than a compliant TCP without trusting any other party, neither the protocol nor the remote STP instance. This seems a strong result. Nonetheless, we expect further work in this area and consider our rate policer to be preliminary. That is because the concept of TCP-friendliness is typically used to compare the performance of specific protocols with that of TCP in an offline fashion. Ours is the first work that uses these concepts for online policing of transfer rates at the flow level as far as we are aware, and adversarial transports are likely to raise further issues. For example, TFRC, which we adapted, originally assumed fixed packet sizes and receiver cooperation, both of which we revisited. Fortunately, TRFC included measures that go beyond classic TCP-friendliness, including limiting growth to be no faster than slow-start at all times. Further measures are likely to be needed. For example, TCP-friendliness governs only long-term transfer rate and does not by itself limit short term dynamics – the choice of averaging interval is left as an implementation decision.

Transport Protocols as a Restricted Domain: Our experience with STP has shown that there are distinct advantages to working in the restricted domain of TCP-friendly transport protocols, rather than adding general-purpose extensibility to an OS kernel. The protocol extensions we considered have access to a fixed, relatively small set of interfaces. The resources used in this domain, such as packet buffers and socket endpoints, all have well-defined semantics and can be managed by the system in a straightforward way. Since protocol extensions are relatively independent of each other and of the rest of the kernel, they are amenable to a simpler protection and termination model than are more generic kernel extensions. Finally, we have a well-defined model for global network safety, discussed above. In sum, the STP domain permitted us to develop a clear model for what it means for a protocol extension to be well-behaved, and also to enforce conformance to the model with low runtime overhead.

7. RELATED WORK

STP leverages ideas and techniques from several research areas: mechanisms for safe code execution, protocol design frameworks, and extensible systems.

7.1 Safety Mechanisms

STP uses Cyclone [32], a type-safe C-like language. We chose Cyclone over other language-based systems like Java and SML because it provides an attractive combination of type-safety, familiarity, and performance. The use of Cyclone also enabled fine-grained interactions between the trusted kernel and untrusted transport extensions. This level of interaction would be difficult to support with alternative technologies such as hardware-based virtual memory; software fault isolation (SFI) [60] might be able to achieve such interaction but to our knowledge that has not been demonstrated in a general way.

OKE [12] is an extensibility framework for the Linux kernel. Our work is complementary to theirs as we focus on the restricted domain of transport protocols where the notion of local and network safety are well defined and much more tractable. We have adapted some of their local safety techniques for use in our implementation.

STP's rate control mechanism is based on TFRC [27], which uses equation 1 to implement a slowly-responding, yet TCP-friendly transport protocol. A contribution of STP beyond this work is to enforce TCP-friendly behavior in the presence of adversarial transport layers. We believe this is the first work to propose and implement a scheme for enforcing network safety.

7.2 Protocol Design Frameworks

The x-kernel [28], Fox project [10], HIPPARCH [15], and Prolac [38] support modular yet high performance protocol implementation. Our work develops and promotes an API for transport protocol implementations, but does not decompose them further. We believe the API to be general and portable to divers operating systems, but demonstrating that remains to be done. We build on safe language technology, Cyclone, but in contrast to all the above work, our emphasis and primary contribution is protecting local and network resources without trusting the code nor the connection peer.

RFC 1263 eloquently makes arguments similar to ours against restriction to backwards-compatible protocol extensions, advocating protocol evolution through multiple protocol versions, defaulting to a common base version. They also argued against from-scratch development of new protocols. Although they also suggested a "protocol backplane," our solution radically differs from what they imagined. Most importantly, STP supports *untrusted mobile* code and provides a way to distribute and upgrade transport protocols. Second, RFC 1263 focused on TCP, whereas STP can support a wide variety of TCP-friendly unicast transport protocols.

7.3 Extensible Systems

Another body of related work is that of extensible OS services, such as the *Plexus* extensible protocol architecture in SPIN [19] and OKE [12]. Much of STP's approach to host safety relates to this work. However, STP tackles the relatively new problems of remote extensibility and guaranteeing TCP-friendly congestion control behavior.

Finally, STP is philosophically similar to active networks [58]. Both systems use mobile code to reduce the lag time for introducing new protocols. The key difference is that STP focuses on transport-layer extensibility, whereas active networks focuses on the network layer. We believe many of the security and deployment concerns that have plagued active network will prove more tractable at the transport level. However, transport-layer extensibility poses new challenges: the extensions must be TCP-friendly, and new transports must integrate seamlessly with unmodified applications.

8. CONCLUSION

Improvements to the transport layer have played a crucial role in the expansion and evolution of the Internet. Unfortunately, since many transport extensions require changes to both endpoints, even simple extensions like SACK require years to reach widespread deployment. In some cases protocol designers have been forced to accept "dumbed down" extensions that only require upgrades at a single site and provide less benefit.

We have presented STP, a novel system that eliminates deployment lag for new transport extensions. STP's approach is to push untrusted transport extensions onto the network endpoints. STP extensions reside beneath the BSD socket layer, and applications require no modifications to use extensions. A key challenge is safety for both host and network resources. STP meets these challenges by leveraging recent research in type-safe languages and robust protocol design.

STP is a working system, and we have ported several transport protocols and extensions, including TCP NewReno, SACK, and a rate-controlled version of UDP. Our evaluation demonstrates that STP enforces limits on host and network resources, while imposing only modest computational overhead and minimal functionality constraints on new transports.

Acknowledgements

We thank the anonymous reviewers whose comments helped improve this paper. We are grateful to Mike Hibler, Eric Eide, Rob Ricci, John Regehr, Kirk Webb, and Ratul Mahajan for their comments and help in evaluation and editing. This work was supported in part by DARPA grants F30602-99-1-0503, F33615-00-C-1696, and F30602-00-2-0565, NSF grant ANI-0082493, and a Microsoft Endowment Fellowship.

9. REFERENCES

[1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the Performance of TCP Pacing. In *Proc. of IEEE INFOCOM*, pages 1157–1165, Mar. 2000.

[2] M. Allman. TCP Congestion Control with Appropriate Byte Counting. RFC 3465, IETF, Feb. 2002.

[3] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042, IETF, Jan. 2001.

[4] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP Over Satellite Channels using standard Mechanisms. RFC 2488, IETF, Jan. 1999.

[5] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, IETF, Apr. 1999.

[6] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 213–226, Oct. 2000.

[7] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, Oct. 2000.

[8] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. of ACM SIGCOMM*, pages 175–187, Sept. 1999.

[9] D. J. Bernstein and E. Schenk. TCP Syn Cookies. 1996, 2002; http://cr.yp.to/syncookies.html.

[10] E. Biagioni. A Structured TCP in Standard ML. In *Proc. of ACM SIGCOMM*, pages 36–45, Aug. 1994.

[11] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. *Software–Practice and Experience*, 25(S4):87–130, Dec. 1995. Also available as Digital Systems Research Center Research Report 115.

[12] H. Bos and B. Samwel. Safe Kernel Programming in the OKE. In *Proc. of the Fifth IEEE Conference on Open Architectures and Network Programming*, June 2002.

[13] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. RFC 1644, IETF, July 1994.

[14] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, Oct. 1995.

[15] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. In *Proc. of ACM SIGCOMM*, pages 60–71, Aug. 1996.

[16] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from HotJava to Netscape and beyond. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[17] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signaling. In *Proc. of the 9th IEEE International Conference on Network Protocols*, pages 332–341, Nov. 2001.

[18] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM Computer Communication Review*, 26(3):5–21, July 1996.

[19] M. E. Fiuczynski and B. N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proc. of the USENIX Annual Technical Conference*, pages 55–64, Jan. 1996.

[20] S. Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, IETF, Apr. 1999.

[21] S. Floyd. Congestion Control Principles. RFC 2914, IETF, Sept. 2000.

[22] S. Floyd. Questions about TCP deployment, 2000. http://www.icir.org/floyd/questions.html.

[23] S. Floyd. HighSpeed TCP for Large Congestion Windows, July 2003. IETF, Internet Draft.

[24] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based Congestion Control for Unicast Applications. In *Proc. of ACM SIGCOMM*, pages 43–56, Aug. 2000.

[25] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883, IETF, July 2000.

[26] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, June 2002.

[27] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, IETF, Jan. 2003.

[28] 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.

[29] V. Jacobson. Congestion Avoidance and Control. In *Proc. of ACM SIGCOMM*, pages 314–329, Aug. 1988.

[30] V. Jacobson. 4BSD TCP Header Prediction. *ACM Computer Communication Review*, 20(2):13–15, Apr. 1990.

[31] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, IETF, May 1992.

[32] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Chene, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proc. of USENIX Annual Technical Conference*, June 2002.

[33] C. Jin, D. X. Wei, and S. H. Low. FAST TCP for High-Speed Long-Distance Networks, June 2003. IETF, Internet Draft.

[34] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 156–171, Dec. 1995.

[35] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained

Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.
[36] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, Nov. 1991.
[37] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram Congestion Control Protocol (DCCP), Oct. 2002. http://www.icir.org/kohler/dccp/.
[38] E. Kohler, F. Kaashoek, and D. Montgomery. A Readable TCP in the Prolac Protocol Language. In *Proc. of ACM SIGCOMM*, pages 3–13, Sept. 1999.
[39] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communication Review*, 30(1):30–36, Jan. 2000.
[40] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *Proc. of ACM SIGCOMM*, pages 281–291, Aug. 1996.
[41] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, IETF, Oct. 1996.
[42] S. O'Malley and L. Peterson. TCP Extensions Considered Harmful. RFC 1263, IETF, Oct. 1991.
[43] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proc. of ACM SIGCOMM*, pages 303–314, Sept. 1998.
[44] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *Proc. of ACM SIGCOMM*, pages 287–298, Aug. 2001.
[45] P. Patel and J. Lepreau. Hybrid Resource Control for Active Extensions. In *Proc. of the Sixth IEEE Conference on Open Architectures and Network Programming*, pages 23–31, Apr. 2003.
[46] P. Patel, D. Wetherall, J. Lepreau, and A. Whitaker. TCP Meets Mobile Code. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems*. IEEE Computer Society, May 2003.
[47] V. Paxson. End-to-end Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.
[48] Pittsburgh Supercomputing Center. The TCP-Friendly Website, 2003. http://www.psc.edu/networking/tcp_friendly.html.
[49] J. Postel. Transmission Control Protocol. RFC 793, IETF, Sept. 1981.
[50] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, Sept. 2001.
[51] K. Ratnam and I. Matta. WTCP: An Efficient Transmission Control Protocol for Networks with Wireless Links. In *Proc. of the Third IEEE Symposium on Computers and Communications*, pages 74–78, June 1998.
[52] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. of IEEE INFOCOM*, pages 1337–1345, Mar. 1999.
[53] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), Jan. 1997.
[54] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proc. of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 155–166, Aug. 2000.
[55] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, IETF, Jan. 1997.
[56] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, Oct. 2000.
[57] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. RFC 3309, IETF, Sept. 2002.
[58] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.
[59] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.
[60] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Dec. 1993.
[61] R. Wang, M. Valla, M. Y. Sanadidi, and M. Gerla. Adaptive Bandwidth Share Estimation in TCP Westwood. In *Proc. of IEEE Globecom*, Nov. 2002.
[62] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. of the 17th ACM Symposium on Operating System Principles*, pages 64–79, Dec. 1999.
[63] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Dec. 2002.
[64] J. Widmer. Equation-based Congestion Control, 2000. Diploma Thesis. University of Mannheim.
[65] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. TR 006, International Computer Science Institute, July 2002.