USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

TCP Meets Mobile Code

Parveen Patel David Wetherall Jay Lepreau Andrew Whitaker
University of Utah and University of Washington

Abstract

This paper argues that transport protocols such as TCP provide a rare domain in which protocol extensibility by untrusted parties is both valuable and practical. TCP continues to be refined despite more than two decades of progress, and the difficulties due to deployment delays and backwards-compatibility are well-known. Remote extensibility, by which a host can ship the transport protocol code and dynamically load it on another node in the network on a per-connection basis, directly tackles both of these problems. At the same time, the unicast transport protocol domain is much narrower than other domains that use mobile code, such as active networking, which helps to make extensibility feasible. The transport level provides a well understood notion of global safety—TCP friendliness—while local safety can be guaranteed by isolation of per-protocol state and use of recent safe-language technologies. We support these arguments by outlining the design of XTCP, our extensible TCP framework.

1 Introduction

TCP was designed over two decades ago and has been evolving ever since. Proposals for changes show no sign of ceasing, as they are driven by changes in the way the network is used and the quest for ever better performance [22, 11, 18, 13, 6, 15, 12, 30, 31, 21, 24, 47, 2, 20, 27, 29, 40, 43, 8, 10, 3, 1, 38, 46, 32, 37]. Yet the process of evolution itself is not simple or painless. Experimentation with a new version of TCP requires that both communication endpoints be upgraded, in the general case. Widespread deployment is needed to unlock the true value of such extensions, which in practice takes years, lowering the value for early adopters and posing a further barrier to change.

These difficulties have resulted in pressure to produce TCP extensions that require upgrades at a single endpoint, even at the expense of efficiency or robustness. For example, both NewReno [18] and SACK [31] improve performance when there are multiple packet losses in one window of data. NewReno is based on a heuristic interpretation of duplicate acknowledgments, and can be deployed for immediate benefit. In contrast, SACK addresses multiple losses by design, has been shown to provide improved performance, and is generally considered the better alternative [17]. The catch is that SACK requires both ends to be upgraded.

An alternative approach is to build remote extensibility mechanisms into TCP itself, so that both end points can be upgraded at once. This would free protocol designers from both the constraints of backwards-compatibility and the deployment barrier. This argument should sound familiar: it is essentially the argument for active networks, which aims to allow new network services to be introduced using mobile code. Yet active networking has not seen widespread deployment; the many reasons include the lack of compelling applications and the technical difficulties of running user-defined code within the network. Furthermore, prior work on extensible operating systems cannot readily be used in this domain due to its lack of support for extending a remote operating system with untrusted protocol code.

In this paper, we put forth the case for XTCP, a remotely extensible version of TCP. We argue that the domain of TCP extensions provides a sweet spot that is well-suited to leverage the mobile code aspect of active networking, without incurring the problems that have hindered active networks. Past and proposed TCP variations demonstrate a clear need for such extensibility: we present an analysis of 27 TCP variants and find that the majority would benefit from remote extensibility. At the same time, compared to the generality of active networks, TCP provides a restricted domain within which the technical challenges can be successfully tackled.

A key challenge in providing this rapid deployment model is to do so without causing security problems. Transport protocol code that comes from sources that are not authoritative—such as the other end of a wide-area connection—should not be trusted. We must ensure that such code cannot compromise the integrity of the host system or consume too many of its resources. Further, standard practice in the networking community is to re-

Author information: Patel, Lepreau: University of Utah, {ppatel,lepreau}@cs.utah.edu; Wetherall, Whitaker: University of Washington, {djw,andrew}@cs.washington.edu.

This work was supported in part by DARPA grants F30602-99-1-0503, F33615-00-C-1696, and a Microsoft Endowment Fellowship.

quire that new transport protocols compete fairly with deployed versions of TCP to ensure that they will not undermine the stability of the network [19]. Thus, to provide a system that is acceptable in practice we must provide this form of network safety. Since extensions to TCP are often undertaken to improve performance, we must also allow new transport extensions to be competitive in performance with hard-coded and manually deployed versions.

Our design makes automatic deployment practical by exploiting TCP's connection phase for in-band signaling of protocol requirements, deferring code distribution to user-mode daemons so that later connections benefit. We use the concept of TCP-friendliness to provide a clear model of network safety, and the recent ECN nonce mechanism [16] to enforce TCP-friendliness without trusting local TCP extensions or any remote parties. To obtain host safety with reasonable impact on performance and the structure of traditional kernels, we exploit TCP's stylized memory allocation and its limited sharing between connections, use a C-like type-safe language [25], and enforce resource limitations. We have focused on TCP to date for concreteness, but expect much of our reasoning to apply to other transports such as DCCP [27] and SCTP [43].

The rest of this paper is organized as follows. In section 2, we develop the case in favor of remote extensibility at the transport layer, while in section 3 we show that such extensibility is achievable by presenting the design of the XTCP framework. In section 4, we conclude with a discussion of the key issues for our future research.

2 The Case for XTCP

2.1 XTCP is Useful

We envision the following scenarios for using XTCP:

1. A "high performance" TCP is installed along with a Web server, and code is pushed to receivers to provide more rapid downloads. Figure 1 illustrates this example scenario using the XTCP extensibility model discussed in section 3.1.

2. A mobile client installs "TCP connection migration" [41] and ships code to the server to allow itself to move.

3. A user installs "TCP nice" [45] to provide background data transfers. No remote code shipping is needed.

To demonstrate that such extensibility is useful, we surveyed TCP extensions and TCP-friendly transports that have been deployed or proposed since congestion control was first introduced in 1988 in TCP Tahoe [22]. We analyzed 27 TCP extensions and classified them into three categories according to which endpoints must be upgraded to gain a benefit, assuming Tahoe as the baseline implementation. The results are shown in Table 1.

We found that the 16 extensions listed in Category 1 require upgrades to both sender and receiver sides to be of value. For some of these extensions, such as TCP connection migration, it is very hard, if not impossible, to benefit by modifying only one endpoint. XTCP provides a clear benefit for these extensions, allowing them to be readily deployed where they otherwise could not.

The five extensions listed in Category 2 can be implemented by upgrading a single endpoint and XTCP would not seem to benefit them directly. However, all of these designs have the potential to become either more robust or effective if both ends can be upgraded and the new functionality split freely between the sender and the receiver. For example, NewReno could become SACK, and TCP Vegas [11] could use receiver timings to more accurately estimate queuing delay [35]. That is, these extensions are constrained to some extent by the pressure of backwards-compatibility; XTCP would alleviate this pressure.

Finally, the remaining six extensions in Category 3 require changes to only one endpoint. For example, both the fast recovery modification to the sender-side TCP and TCP-Nice are transparent to the receiver. For these protocols, XTCP can still provide a useful kernel upgrade mechanism by allowing third-party software authors to write and remotely install TCP extensions.

In summary, the majority of the extensions we studied benefit from the XTCP model of remote extensibility, and many would be difficult to deploy without it.

2.2 XTCP is Practical

The second issue we consider is technical feasibility, since XTCP requires the use of mobile code and is similar in spirit to the challenging domain of active networks [44]. The key insight and difference between XTCP and active networks is that TCP (or more generally, unicast transport protocols) is a much narrower domain in which to provide extensibility. This enables several key simplifications that increase our confidence in being able to build an effective solution.

First, XTCP provides extensibility at a much coarser granularity than active networks: per connection at endpoints rather than per packet at routers. This permits a simpler approach to upgrades, where extensions are signaled in-band, at connection setup time, and code is transferred in the background.

Second, there exists a clear model of "global" or network safety for XTCP: a connection should not be able to send faster than a TCP-friendly transport. In fully general active networks, there is no clear limit to

Category	Extensions
1. *Require* both endpoints to change	**1. Connection migration**: Migrating live TCP connections [41], **2. SACK**: Selective acks [31], **3. D-SACK**: Duplicate SACK [21], **4. FACK**: Forward acks [30], **5. RFC 1323**: TCP extensions for high-speed networks [24], **6. TCPSAT**: TCP for satellite networks [4], **7. ECN**: Explicit congestion notification [36], **8. ECN nonce**: Detects masking of ECN signals by the receiver or network [16], **9. RR-TCP**: Robustly handles packet reordering [47], **10. WTCP**: TCP for wireless WANs [37], **11. The Eifel algorithm**: Detection of spurious retransmissions [29], **12. T/TCP**: TCP for transactions [10], **13. TFRC**: Equation-based TCP-friendly congestion control [20], **14. DCCP**: New transport protocol with pluggable congestion control [27], **15. SCTP**: Transport protocol support for multi-homing, multiple streams etc., between endpoints [43], **16. RAP**: Rate adaptive TCP-friendly congestion control [38]
2. *Could benefit more* if both endpoints could change	**1. NewReno**: Approximation of SACK from sender side [18] **2. TCP Vegas**: A measurement-based adaptive congestion control [11], **3. TCP Westwood**: Congestion control using end-to-end rate estimation [46], **4. Karn/Partridge algorithm**: Retransmission backoff and avoids spurious RTO estimates due to retransmission ambiguity [26], **5. Congestion manager**: A generic congestion control layer [7]
3. *Require* only one endpoint to change	**1. Header prediction**: Common case optimization on input path [23], **2. Fast recovery**: Faster recovery from losses [42], **3. Syn-cookies**: Protection against SYN-attacks [8], **4. Limited transmit**: Performance enhancement for lossy networks [2], **5. Appropriate byte-counting**: Counting bytes instead of segments for congestion control [1], **6. TCP nice**: TCP for background transfers [45]

Table 1: Classification of TCP extensions, assuming TCP Tahoe [22] as the baseline version.

the network-wide resources that can be expended on a packet, and even if there were, multiple extensible routers would need to cooperate to enforce the limit. Furthermore, recent advances in network safety mechanisms allow enforcement of a rate-limiting model. Compliance can be checked by the local XTCP using a variant of the ECN nonce mechanism, without trusting either extension code or remote hosts.

Third, even in terms of local safety—protecting the resources of the local host—XTCP affords simplifications compared to active networks. There is limited sharing between TCP connections in the kernel, which translates into simpler protection models and eases the task of termination, should code need to be unloaded for any reason. Recent advances in safe language technology also contribute to XTCP's practicality. Our design leverages Cyclone [25], a type-safe variant of C, to obtain host protection while providing relatively straightforward reuse of existing C-based transport protocols, familiarity to system programmers, and acceptable performance.

A fourth simplification is that XTCP can leverage a large body of past TCP extensions upon which to base its extension API. Active networks, on the other hand, had no such agreed set of applications and hence it tended towards generality. In contrast, XTCP aims to do one thing, and to do it well.

3 Design

The basic approach of XTCP is to download transport extensions directly into the operating system kernel. To guarantee safety, XTCP uses type-safety to achieve memory protection and restricts extensions to a resource-safe API, called the XTCP API, summarized in Table 2. Extensions are invoked in response to specific system events, such as packet input/output and timers, and are allowed to read and write IP datagrams. To ensure network safety, trusted XTCP network access functions attach IP headers to outgoing datagrams and limit the sending rate of a transport to that of a TCP-friendly transport. Extensions are allowed to register timers, manipulate packet buffers, and interact with the sockets layer in a safe manner. This functionality is sufficient to implement a range of transport protocols, including conventional TCP. In this section, we focus on three key aspects of XTCP's design: *connection setup and code distribution*, *network safety*, and *host safety*.

3.1 Connection Setup and Code Distribution

Before a transport extension can be used, the code must be distributed to both connection endpoints. XTCP accomplishes this by interposing on normal TCP connection setup, as shown in Figure 1. Suppose host A wants to communicate with host B using a transport extension. Host A's first packet (either a SYN or a SYN-ACK) includes a special TCP option, which includes a hash of the desired transport extension's code. If host B has already loaded the extension, the current connection is established using the requested protocol. If B has not loaded the extension, it issues a request to A for the new protocol code, while using the default TCP to establish the current connection.

Asynchronously "at leisure," a user-level daemon on A transfers the code to B's daemon (connection 2 in the figure). B either compiles source code, requests a trusted server to do so, or verifies the authenticity of object code,

The XTCP API Categories
1. Protocol management
xtcp_load_proto(proto_sw)
xtcp_unload_proto(proto_handle)
2. Sockets layer
xtcp_sowakeup(socket)
xtcp_sbapppend(socket, seg)
xtcp_isdisconnecting(socket)
xtcp_sobind(socket)
3. Connection management callback functions
xtcp_attach(socket)
xtcp_connect(socket, remote-endpoint, state)
xtcp_abort(socket)
xtcp_accept(socket, remote-endpoint, state)
4. TCP-friendly network access
xtcp_ack(end_seqno, nonce)
xtcp_ack_sum(end_seqno, noncesum)
xtcp_nack(seqno)
xtcp_net_sendack(segment)
xtcp_net_send(segment, seqno)
xtcp_net_resend(segment, new_seqno, old_seqno)
5. Runtime support
xtcp_gettick()
xtcp_seg_alloc(proto_handle)
xtcp_timer_reset(proto_handle, callout)
xtcp_get_rtentry(proto_handle, dst_ip_addr)

Table 2: The initial XTCP API exports a set of 67 functions. The major groups and sample functions are listed.

loads it into the kernel, and makes it available for subsequent network connections. This signaling and code distribution scheme only benefits later connections—often substantially later, since the entire process could take many seconds or perhaps even minutes, since it must be done at low priority to mitigate DoS attacks. Its effectiveness relies on the pattern of TCP connections common in today's Internet, in which hosts make repeated TCP connections to a particular peer.

This connection setup procedure has several features we believe are important to practical deployment. It is fully backwards-compatible with conventional TCP, imposes minimal control latency on the connection that is used to bootstrap a new extension, minimally invades the kernel software architecture, and retains TCP's three-way packet exchange for compatibility with existing level 4 "middleboxes" like firewalls, NAT boxes, and other proxies. However, note that this backwards-compatible procedure is limited to extensions that use TCP-like handshakes; new transport protocols are free to use their own connection protocol after bootstrapping.

Our current design provides extensibility at the granularity of an entire TCP implementation. This coarse-grained design provides complete flexibility and is practical in most domains: our measurements show that a full TCP implementation, with comments and headers, takes 85K bytes of compressed source code; Cyclone source code should not expand it at all [25]. If certified object code is transferred instead of source, its size is smaller. We measured 20K of x86 object code, which Cyclone should expand by at most 20%.

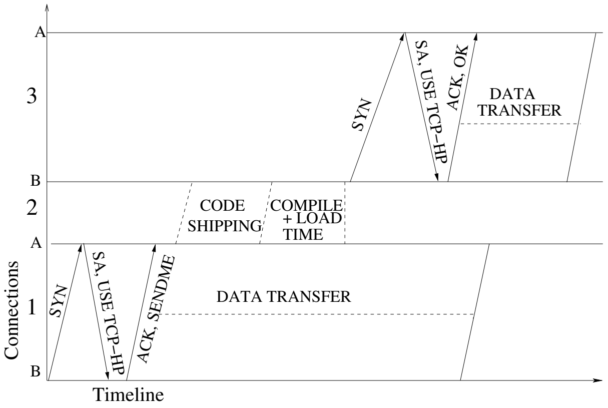

Figure 1: An example scenario. In connection 1, server A requests use of high-performance TCP (TCP-HP), causing client B to ask for the code. At application level in connection 2, server A sends it to client B. In a later, unrelated connection 3, A and B use TCP-HP.

We have not discussed the multitude of potential policy issues regarding which extensions to invoke or accept from peers. Briefly, our design uses three sources to select extensions: application-provided socket options, host-wide configuration options, and the code distribution and policy server that communicates with other such servers. The policy servers might "rate" protocols [39], could form a web of (partial) trust, and in fact could represent the beginnings of an Internet "knowledge plane" [14]. However, we believe only the simplest policies need be implemented for our design to function well.

3.2 Network Safety

The network safety goal of XTCP is to require new transport protocols to compete fairly with deployed versions of TCP to ensure that they will not undermine the stability of the network, as recommended in RFC 2914 [19]. Currently, XTCP achieves this by limiting extensions to a TCP-friendly sending rate, as modeled by the TCP rate equation [33]. This equation gives an upper bound on allowable sending rate of a TCP-friendly flow in terms of *packet size*, *loss event rate* and *round trip time*.

XTCP must compute values of these parameters without trusting the local transport or remote endpoint. This is essential to prevent new transports from compromising the rate-checking mechanism by indirectly inflating the allowable sending rate. For this purpose, XTCP adapts the recently proposed ECN nonce mechanism [16].

The ECN nonce mechanism is based on placing a random one-bit value in the IP header of outgoing packets. The nonce bit (or a sum of nonce bits over many packets) is later used as a proof of acknowledgment. The extensions must inform XTCP of packet arrival and loss events in terms of per-packet sequence numbers; these numbers appear as an extra argument to the network send function. Upon receiving an acknowledgment, the extension reports the sequence number and nonce, using one of the acknowledgment functions shown in Table 2. Packet drops are indicated by using negative-acknowledgment functions; also, XTCP assumes that a packet has been lost if the nonce is incorrect or a timeout period has expired. This information is sufficient for XTCP to estimate the packet size, loss event rate, and round trip time parameters needed by the TCP rate equation.

A crucial feature of the above mechanism is that XTCP remains independent of transport header formats, permitting arbitrary header modifications by new extensions.

3.3 Host Safety

Host safety in the face of untrusted code is achieved through principles of isolation and resource control similar to those used in safe language-based operating systems, such as KaffeOS [5], but simplified by the constrained memory allocation and sharing behavior of TCP. Memory safety is achieved by using Cyclone, a type-safe dialect of C [25]. The type-safety of Cyclone prevents memory corruption, and its compatibility with C makes it easier and more efficient to interface with traditional kernels than other safe languages, such as Java or OCaml.

Extensions are never allowed to share memory with other extensions, grab system locks, or disable interrupts. Therefore, asynchronous termination can be safely achieved by terminating all connections of a misbehaving extension. This tractable notion of termination allows us to use traditional run-time techniques to bound memory usage and inexpensive timer interrupt-based checking to bound CPU usage.

4 Open Issues and Conclusion

In this paper, we have argued that TCP, and more generally unicast transport protocols, present a unique domain in which remote extensibility by untrusted parties is both valuable for users and technically feasible. We presented the design of XTCP, our framework for achieving this extensibility. We have implemented a prototype of XTCP in the FreeBSD kernel and ported TCP NewReno [18] and TCP SACK [31] to it. Our initial experience with XTCP's performance, safety characteristics, and ease of porting existing protocols has been encouraging [34].

There are several open issues that we expect to tackle as we gain experience with the system. First, we are using the TCP-friendly rate equation to provide network safety. This equation governs the steady-state transfer rate as a function of loss, and to date it has mostly been used to build other transports such as TFRC. We use it online to police TCP extensions. This means we must determine appropriate timescales on which to apply it. The timescales must be long enough to avoid false alarms, but short enough to prevent abusive transports from crowding out compliant transports.

Second, a key issue is whether our XTCP API will prove sufficient to support a wide variety of TCP extensions, including those currently unimagined. The need to repeatedly revise the XTCP API would defeat its purpose. We believe that our static API in conjunction with mobile code will prove sufficiently general because it fulfills a key need of extensions: the ability to change packet formats, allowing the sender and receiver to exchange and process new information. Our current API directly supports 18 of the 21 extensions listed in Categories 1 and 2 in table 1. Three extensions cannot be supported because they are not TCP-friendly. However, only experience will tell whether XTCP can fulfill its twin goals of supporting a large fraction of useful transport extensions while guaranteeing host and network safety.

A related issue is whether we can design an OS-independent XTCP API such that the same code can be compiled across operating systems. This will further expedite protocol deployment by allowing developers to "write-once" and bypass all OS-specific issues.

Finally, it is interesting to consider the granularity of extensions. We began by partially designing a fine-grained extensibility model, but switched to shipping TCP implementations in their entirety. This coarse-grained model provides complete flexibility, avoids feature interaction, is simple, and appears practical, given the observed modest code sizes. A fine-grained composable TCP implementation, analogous to that in Prolac [28] and FoxNet [9], would reduce the size of transported code and host memory consumption, but could lead to a less flexible extension model. We suspect that simple will win.

Acknowledgments

We thank the anonymous reviewers whose comments helped improve an earlier version of this paper. We are grateful to Tim Stack, Mike Hibler, Rob Ricci, and John Regehr for implementation, evaluation, and editing help.

References

[1] M. Allman. TCP Congestion Control with Appropriate Byte Counting. *IETF, Internet Draft, draft-allman-tcp-abc-04.txt*, Oc-

tober 2002.

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

[3] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. *RFC 3390*, 2002.

[4] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP Over Satellite Channels using Standard Mechanisms. *RFC 2488*, January 1999.

[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *OSDI*. USENIX Association, Oct. 2000.

[6] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. *15th International Conference on Distributed Computing Systems*, 1995.

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

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

[9] E. Biagioni. A Structured TCP in Standard ML. In *ACM SIGCOMM*, 1994.

[10] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. *RFC 1644*, 1994.

[11] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE JSAC*, 13(8), 1995.

[12] K. Brown and S. Singh. M-TCP: TCP for Mobile Cellular Networks. *Computer Communication Review*, 1997.

[13] F. Chengpeng. *TCP Veno: End-To-End Congestion Control Over Heterogeneous Networks*. PhD thesis, Chinese University of Hong Kong, 2001.

[14] D. D. Clark, C. Partridge, and J. C. Ramming. A Knowledge Plane for the Internet, Feb. 2003. Unpublished report; earlier versions available at http://www.isi.edu/braden/know-plane/.

[15] R. Durst, G. Miller, and E. Travis. TCP Extensions for Space Communications. In *ACM MobiCom*, Nov. 1996.

[16] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signaling. In *IEEE ICNP*, November 2001.

[17] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM Computer Communication Review*, 26(3), Jul 1996.

[18] S. Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 2582*, 1999.

[19] S. Floyd. Congestion Control Principles. *RFC 2914*, September 2000.

[20] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.

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

[22] V. Jacobson. Congestion Avoidance and Control. *SIGCOMM*, 1988.

[23] V. Jacobson. 4BSD Header Prediction. *ACM Computer Communication Review*, April 1990.

[24] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC 1323*, 1992.

[25] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Chene, and Y. Wang. Cyclone: A Safe Dialect of C. *USENIX Annual Techincal conference, Monterey, CA*, June 2002.

[26] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, 1991.

[27] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram Congestion Control Protocol (DCCP), October 2002. http://www.icir.org/kohler/dccp/.

[28] E. Kohler, F. Kaashoek, and D. Montgomery. A Readable TCP in the Prolac Protocol Language. In *SIGCOMM*, pages 3–13, 1999.

[29] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communication Review*, 30(1), January 2000.

[30] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *SIGCOMM*, 1996.

[31] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. *RFC 2018*, 1996.

[32] J. Nagle. Congestion Control in IP/TCP. *RFC 896*, January 1984.

[33] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM*, 1998.

[34] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading Transport Protocols using Untrusted Mobile Code. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003. To appear.

[35] V. Paxson. End-to-end Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3), 1999.

[36] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168*, 2001.

[37] K. Ratnam and I. Matta. WTCP: An Efficient Transmission Control Protocol for Networks with Wireless Links. *Proc. Third IEEE ISCC*, 1998.

[38] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet. In *INFOCOM (3)*, 1999.

[39] R. Ricci and J. Lepreau. Active Protocols for Agile Censor-Resistant Networks. In *8th HotOS*. IEEE Computer Society, May 2001.

[40] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *ACM Computer Communication Review*, 28(4), October 1998.

[41] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *6th MobiCom*, 2000.

[42] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *RFC 2001*, January 1997.

[43] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. *RFC 3309*, 2002.

[44] 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, 1997.

[45] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *OSDI*, 2002.

[46] R. Wang, M. Valla, M. Y. Sanadidi, and M. Gerla. Adaptive Bandwidth Share Estimation in TCP Westwood. In *IEEE Globecom*, November 2002.

[47] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. TR 006, International Computer Science Institute (ICSI), Berkeley, California, USA, July 2002.