Fitting Square Pegs Through Round Pipes

Unordered Delivery Wire-Compatible with TCP and TLS

Michael F. Nowlan[†] Nabin Tiwari[*] Janardhan Iyengar[*] Syed Obaid Amin[†] Bryan Ford[†]

Abstract

Internet applications increasingly employ TCP not as a *stream abstraction*, but as a *substrate* for application-level transports, a use that converts TCP's in-order semantics from a convenience blessing to a performance curse. As Internet evolution makes TCP's use as a substrate likely to grow, we offer *Minion*, an architecture for backward-compatible out-of-order delivery atop TCP and TLS. Small OS API extensions allow applications to manage TCP's send buffer and to receive TCP segments out-of-order. Atop these extensions, Minion builds application-level protocols offering true unordered datagram delivery, within streams preserving strict wire-compatibility with unsecured or TLS-secured TCP connections. Minion's protocols can run on unmodified TCP stacks, but benefit incrementally when either endpoint is upgraded, for a backward-compatible deployment path. Experiments suggest that Minion can noticeably improve performance of applications such as conferencing, virtual private networking, and web browsing, while incurring minimal CPU or bandwidth costs.

1 Introduction

TCP [46] was originally designed to offer applications a convenient, high-level communication abstraction with semantics emulating Unix file I/O or pipes. As the Internet has evolved, however, TCP's original role of offering an *abstraction* has gradually been supplanted with a new role of providing a *substrate* for transport-like, application-level protocols such as SSL/TLS [17], ØMQ [3], SPDY [2], and WebSockets [52]. In this new *substrate* role, TCP's in-order delivery offers little value since application libraries are equally capable of implementing convenient abstractions. TCP's strict in-order delivery, however, prevents applications from controlling the *framing* of their communications [14, 19], and incurs a "latency tax" on content whose delivery must wait for the retransmission of a single lost TCP segment.

Due to the difficulty of deploying new transports today [20, 36, 41], applications rarely utilize new out-of-order transports such as SCTP [45] and DCCP [28]. UDP [37] is a popular substrate, but is still not universally supported in the Internet, leading even delay-sensitive applications such as the Skype telephony system [7] and Microsoft's DirectAccess VPN [16], to fall back on TCP despite its drawbacks.

Recognizing that TCP's use as a substrate is likely to continue and expand, we introduce *Minion*, a novel architecture for efficient but backward-compatible unordered delivery in TCP. Minion consists of u*TCP*, a small OS extension adding basic unordered delivery primitives to TCP, and two application-level protocols implementing datagram-oriented delivery services that function on either uTCP or unmodified TCP stacks.

uTCP addresses delays caused by TCP's send and receive buffering. On the send side, uTCP gives the application a controlled ability to insert data out-of-order into TCP's send queue, allowing fresh high-priority data to bypass previously-queued low-priority data, for example. On the receive side, uTCP enables the application to receive out-of-order TCP segments immediately, without delaying their delivery until retransmissions fill prior holes. Designed for simplicity and deployability, these extensions add less than 600 lines to Linux's TCP stack.

Minion's application-level protocols, u*COBS* and u*TLS*, build general datagram delivery services atop uTCP or TCP. Key challenges these protocols address are: (a) TCP offers no reliable out-of-band framing to delimit datagrams in a TCP stream; (b) uTCP cannot add out-of-band framing without changing TCP's wire protocol; and (c) common in-band TCP framing methods assume in-order processing. To make datagrams *self-delimiting* in a TCP stream, uCOBS leverages *Consistent Overhead Byte Stuffing* (COBS) [12] to encode application datagrams with at most 0.4% expansion, while reserving a single byte value to delimit encoded datagrams.

Minion adapts the stream-oriented TLS [17] into a secure datagram delivery service atop uTCP or TCP. To avoid changing the TLS wire protocol, the uTLS receiver heuristically "guesses" TLS record boundaries in stream fragments received out-of-order, then leverages TLS's cryptographic MAC to confirm these guesses reliably. By preserving strict wire-compatibility with TLS, uTLS enables unordered delivery within streams indistinguishable in the network from HTTPS [39], for example.

Experiments with a prototype on Linux show several benefits for applications using TCP. Minion can reduce application-perceived jitter of Voice-over-IP (VoIP) streams atop TCP, and increase perceptible-quality metrics [32]. Virtual private networks (VPNs) that tunnel IP traffic over SSL/TLS, such as OpenVPN [34] or Di-

[*] Franklin and Marshall College

[†] Yale University

rectAccess [16], can double the throughput of some tunneled TCP connections, by employing uTCP to prioritize and expedite tunneled ACKs. Web transports can cut the time before a page begins to appear by up to half, achieving the latency benefits of multistreaming transports [19, 31] while preserving the TCP substrate. Use of uCOBS can incur up to $5\times$ CPU load with respect to raw TCP, due to COBS encoding, but for secure connections, uTLS incurs less than 7% CPU overhead (and no bandwidth overhead) atop the baseline cost of TLS 1.1.

This paper's primary contributions are: (a) the first wire-compatible TCP extension we are aware of offering true out-of-order delivery; (b) an API allowing applications to prioritize TCP's send queue; (c) a novel use of COBS [12] for out-of-order framing atop TCP; (d) an existence proof that out-of-order datagram delivery is achievable from the unmodified, stream-based TLS wire protocol; (e) a prototype and experiments demonstrating Minion's practicality and performance benefits.

Section 2 motivates Minion by discussing the evolution of TCP's role in the Internet. Section 3 introduces Minion's high-level architecture, and Section 4 describes its uTCP extensions. Section 5 presents uCOBS for non-secure datagram delivery, and Section 6 details uTLS, a secure analog. Section 7 discusses the current Minion prototype, and Section 8 evaluates its performance experimentally. Section 9 summarizes related work, and Section 10 concludes.

2 Motivating Minion

This section describes how TCP's role in the network has evolved from a communication *abstraction* to a communication *substrate*, why its in-order delivery model makes TCP a poor substrate, and why other OS-level transports have failed to replace TCP in this role.

2.1 Rise of Application-Level Transports

The transport layer's traditional role in a network stack is to build high-level communication abstractions convenient to applications, atop the network layer's basic packet delivery service. TCP's reliable, stream-oriented design [46] exemplified this principle, by offering an inter-host communication abstraction modeled on Unix pipes, which were the standard *intra-host* communication abstraction at the time of TCP's design. The Unix tradition of implementing TCP in the OS kernel offered further convenience, allowing much application code to ignore the difference between an open disk file, an intra-host pipe, or an inter-host TCP socket.

Instead of building directly atop traditional OS-level transports such as TCP or UDP, however, today's applications frequently introduce additional transport-like protocol layers at user-level, typically implemented via application-linked libraries. Examples include the ubiquitous SSL/TLS [17], media transports such as RTP [44], and experimental multi-streaming transports such as SST [19], SPDY [2], and ØMQ [3]. Applications increasingly use HTTP or HTTPS over TCP as a substrate [36]; this is also illustrated by the W3C's WebSocket interface [52], which offers general bidirectional communication between browser-based applications and Web servers atop HTTP and HTTPS.

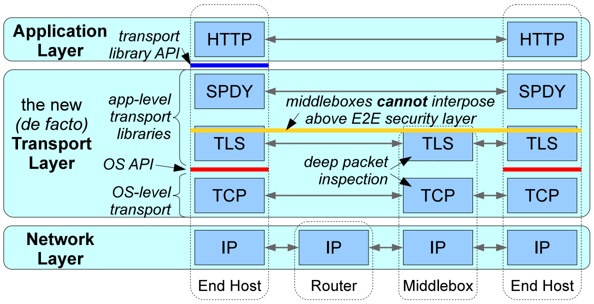


Figure 1: Today's "*de facto* transport layer" is effectively split between OS and application code.

In this increasingly common design pattern, the "transport layer" as a whole has in effect become a stack of protocols straddling the OS/application boundary. Figure 1 illustrates one example stack, representing Google's experimental Chromium browser, which inserts SPDY for multi-streaming and TLS for security at application level, atop the OS-level TCP.

One can debate whether a given application-level protocol fits some definition of "transport" functionality. The important point, however, is that today's applications no longer need, or expect, the underlying OS to provide "convenient" communication abstractions: the application simply links in libraries, frameworks, or middleware offering the abstractions it desires. What today's applications need from the OS is not convenience, but *an efficient substrate* atop which application-level libraries can build the desired abstractions.

2.2 TCP's Latency Tax

While TCP has proven to be a popular substrate for application-level transports, using TCP in this role converts its delivery model from a blessing into a curse. Application-level transports are just as capable as the kernel of sequencing and reassembling packets into a logical data unit or "frame" [14]. By delaying any segment's delivery to the application until all prior segments are received and delivered, however, TCP imposes a "latency tax" on all segments arriving within one round-trip time (RTT) after any single lost segment.

This latency tax is a fundamental byproduct of TCP's in-order delivery model, and is irreducible, in that an application-level transport cannot "claw back" the time a potentially useful segment has wasted in TCP's buffers.

The best the application can do is simply to *expect* higher latencies to be common. A conferencing application can use a longer jitter buffer, for example, at the cost of increasing user-perceptible lag. Network hardware advances are unlikely to address this issue, since TCP's latency tax depends on RTT, which is lower-bounded by the speed of light for long-distance communications.

2.3 Alternative OS-level Transports

All standardized OS-level transports since TCP, including UDP [37], RDP [51], DCCP [28], and SCTP [45], support out-of-order delivery. The Internet's evolution has created strong barriers against the widespread deployment of new transports other than the original TCP and UDP, however. These barriers are detailed elsewhere [20,36,41], but we summarize two key issues here.

First, adding or enhancing a "native" transport built atop IP involves modifying popular OSes, effectively increasing the bar for widespread deployment and making it more difficult to evolve transport functionality below the red line representing the OS API in Figure 1. Second, the Internet's original "dumb network" design, in which routers that "see" only up to the IP layer, has evolved into a "smart network" in which pervasive middleboxes perform deep packet inspection and interposition in transport and higher layers. Firewalls tend to block "anything unfamiliar" for security reasons, and Network Address Translators (NATs) rewrite the port number in the transport header, making both incapable of allowing traffic from a new transport without explicit support for that transport. Any packet content not protected by end-to-end security such as TLS—the yellow line in Figure 1—has become "fair game" for middleboxes to inspect and interpose on [38], making it more difficult to evolve transport functionality anywhere below that line.

2.4 Why Not UDP?

As the only widely-supported transport with out-of-order delivery, UDP offers a natural substrate for application-level transports. Even applications otherwise well-suited to UDP's delivery model often favor TCP as a substrate, however. A recent study found over 70% of streaming media using TCP [23], and even latency-sensitive conferencing applications such as Skype often use TCP [7].

Network middleboxes support UDP widely but not *universally*. For this reason, latency-sensitive applications seeking maximal connectivity "in the wild" often fall back to TCP when UDP connectivity fails. Skype [7] and Microsoft's DirectAccess VPN [16], for example, support UDP but can masquerade as HTTP or HTTPS streams atop TCP when required for connectivity.

TCP can offer performance advantages over UDP as well. For applications requiring congestion control, an OS-level implementation in TCP may be more timing-accurate than an application-level implementation in a UDP-based protocol, because the OS kernel can avoid the timing artifacts of system calls and process scheduling [54]. Hardware TCP offload engines can optimize common-case efficiency in end hosts [30], and performance enhancing proxies can optimize TCP throughput across diverse networks [11, 13]. Since middleboxes can track TCP's state machine, they impose much longer idle timeouts on open TCP connections—nominally two hours [22]—whereas UDP-based applications must send keepalives every two minutes to keep an idle connection open [5], draining power on mobile devices.

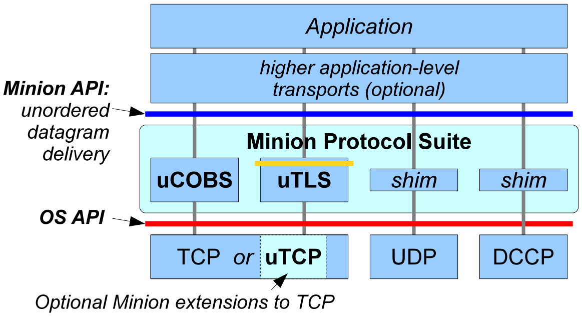

Application
higher application-level transports (optional)
Minion API:
unordered datagram delivery
Minion Protocol Suite
uCOBS
uTLS
shim
shim
OS API
TCP or uTCP
UDP
DCCP
Optional Minion extensions to TCP

Figure 2: Minion architecture

For applications, TCP versus UDP represents an "all-or-nothing" choice on the spectrum of services applications need. Applications desiring some but not all of TCP's services, such as congestion control but unordered delivery, must reimplement and tune all other services atop UDP or suffer TCP's performance penalties.

Without dismissing UDP's usefulness as a truly "least-common-denominator" substrate, we believe the factors above suggest that TCP will also remain a popular substrate—even for latency-sensitive applications that can benefit from out-of-order delivery—and that a deployable, backward-compatible workaround to TCP's latency tax can significantly benefit such applications.

3 Minion Architecture Overview

Minion is an architecture and protocol suite designed to meet the needs of today's applications for efficient unordered delivery built atop either TCP or UDP. Minion itself offers no high-level abstractions: its goal is to serve applications and higher application-level transports, by acting as a "packhorse" carrying raw datagrams as reliably and efficiently as possible across today's diverse and change-averse Internet.

3.1 All-Terrain Unordered Delivery

Figure 2 illustrates Minion's architecture. Applications and higher application-level transports link in and use Minion in the same way as they already use existing application-level transports such as DTLS [40], the datagram-oriented analog of SSL/TLS [17]. In contrast with DTLS's goal of layering security atop datagram transports such as UDP or DCCP, Minion's goal is to offer efficient datagram delivery atop *any* available OS-level substrate, including TCP.

While many protocols embed datagrams or application-level frames into TCP streams using delimiting schemes, to our knowledge Minion is the first application-level transport that, under suitable conditions, offers true unordered delivery atop TCP. Minion effectively offers relief from TCP's latency tax: the loss of one TCP segment in the network no longer prevents datagrams embedded in subsequent TCP segments from being delivered promptly to the application.

3.2 Minion Architecture Components

Minion consists of several application-level transport protocols, together with a set of optional enhancements to end hosts' OS-level TCP implementations.

Minion's enhanced OS-level TCP stack, which we call *uTCP* ("unordered TCP"), includes sender- and receiver-side API features supporting unordered delivery and prioritization, detailed in Section 4. These enhancements affect only the OS API through which application-level transports such as Minion interact with the TCP stack, and make *no* changes to TCP's wire protocol.

Minion's application-level protocol suite currently consists of uCOBS, which implements unordered datagram delivery atop unmodified TCP or uTCP streams using COBS encoding [12] as described in Section 5; and uTLS, which adapts the traditionally stream-oriented TLS [17] into a secure unordered datagram delivery service atop TCP or uTCP. Minion also adds trivial shim layers atop OS-level datagram transports, such as UDP and DCCP, to give applications a consistent API for unordered delivery across multiple OS-level transports.

Minion currently leaves to the application the decision of *which* protocol to use for a given connection: e.g., uCOBS or uTLS atop TCP/uTCP, or OS-level UDP or DCCP via Minion's shims. We are developing an experimental *negotiation protocol* to explore the protocol configuration space dynamically, optimizing protocol selection and configuration for the application's needs and the network's constraints [21], but we defer this enhancement to future work. Many applications already incorporate simple negotiation schemes, however—e.g., attempting a UDP connection first and falling back to TCP if that fails—and adapting these mechanisms to engage Minion's protocols according to application-defined preferences and decision criteria should be straightforward.

3.3 Compatibility and Deployability

Minion addresses the key barriers to transport evolution, outlined in Section 2.3, by creating a backward-compatible, incrementally deployable substrate for new application-layer transports desiring unordered delivery. Minion's deployability rests on the fact that it can, when necessary, avoid relying on changes either "below the red line" in the end hosts (the OS API in Figure 1), or "below the yellow line" in the network (the end-to-end security layer in Figure 1).

While Minion's uCOBS and uTLS protocols offer maximum performance benefits from out-of-order delivery when both endpoints include OS support for Minion's uTCP enhancements, uCOBS and uTLS still function and interoperate correctly even if neither endpoint supports uTCP, and the application need not know or care whether the underlying OS supports uTCP. If only one endpoint OS supports uTCP, Minion still offers incremental performance benefits, since uTCP's sender-side and receiver-side enhancements are independent. A uCOBS or uTLS connection atop a mixed TCP/uTCP endpoint-pair benefits from uTCP's sender-side enhancements for datagrams sent by the uTCP endpoint, and the connection benefits from uTCP's receiver-side enhancements for datagrams arriving at the uTCP host.

Addressing the challenge of network-compatibility with middleboxes that filter new OS-level transports and sometimes UDP, Minion offers application-level transports a continuum of substrates representing different tradeoffs between suitability to the application's needs and compatibility with the network. An application can use unordered OS-level transports such as UDP, DCCP [28], or SCTP [45], for paths on which they operate, but Minion offers an unordered delivery alternative usable even when TCP is the only viable choice.

A final issue is compatibility with existing applications. Since most of Minion operates at application-level, applications must be changed to use the Minion API. A pair of application endpoints may also need to negotiate whether to use Minion, or to run directly atop OS-level transports for compatibility with earlier versions of the application. This challenge is comparable to the cost of adding TLS or DTLS support to an application, and the popularity of application-level transports such as TLS suggests that these costs are surmountable. Minion's application-level functionality might eventually be merged into existing or future application-level transports and communication frameworks, making its benefits available with few or no application changes.

4 uTCP: Unordered TCP

Minion enhances the OS's TCP stack with API enhancements supporting unordered delivery in both TCP's send and receive paths, enabling applications to reduce transmission latency at both the sender- and receiver-side end hosts when both endpoints support uTCP. Since uTCP makes no changes to TCP's wire protocol, two endpoints need not "agree" on whether to use uTCP: one endpoint gains latency benefits from uTCP even if the other endpoint does not support it. Further, an OS may choose independently whether to support the sender- and receiver-side enhancements, and when available, applications can activate them independently.

In this spirit of Section 2, uTCP does *not* seek to offer "convenient" or "clean" unordered delivery abstractions directly at the OS API. Instead, uTCP's design is motivated by the goals of maintaining exact compatibility with TCP's existing wire-visible protocol and behavior, and facilitating deployability by minimizing the extent and complexity of changes to the OS's TCP stack. The design presented here is only one of many viable approaches, with different tradeoffs, to supporting unordered delivery in TCP. Section 4.3 briefly outlines a few such alternatives.

We describe uTCP's API enhancements in terms of the BSD sockets API, although uTCP's design contains nothing inherently specific to this API.

4.1 Receiver-Side Enhancements

uTCP adds one new socket option affecting TCP's receive path, enabling applications to request immediate delivery of TCP segments received out of order. An application opens a TCP stream the usual way, via `connect()` or `accept()`, and may use this stream for conventional in-order communication before enabling uTCP. Once the application is ready to receive out-of-order data, it enables the new option `SO_UNORDERED` via `setsockopt()`, which changes TCP's receive-side behavior in two ways.

First, whereas a conventional TCP stack delivers received data to the application only when prior gaps in the TCP sequence space are filled, the uTCP receiver makes data segments available to the application immediately upon receipt, skipping TCP's usual reordering queue. The application obtains this data via `read()` as usual, but the first data byte returned by a `read()` call may no longer be the one logically following the last byte returned by the prior `read()` call, in the byte stream transmitted by the sender. The data the uTCP stack delivers to the application in successive `read()` calls may skip forward and backward in the transmitted byte stream, and uTCP may even deliver portions of the transmitted stream multiple times. uTCP guarantees only that the data returned by one `read()` call corresponds to *some* contiguous sequence of bytes in the sender's transmitted stream, and that barring connection failure, uTCP will *eventually* deliver every byte of the transmitted stream at least once.

Second, when servicing an application's `read()` call, the uTCP receiver prepends a short header to the returned data, indicating the logical offset of the first returned byte in the sender's original byte stream. The uTCP stack computes this logical offset simply by subtracting the Initial Sequence Number (ISN) of the received stream from the TCP sequence number of the segment being delivered. Using this metadata, the application can piece together data segments from successive `read()` calls into longer contiguous *fragments* of the transmitted byte stream.

Figure 3 illustrates uTCP's receive-side behavior, in a simple scenario where three TCP segments arrive in succession: first an in-order segment, then an out-of-order segment, and finally a segment filling the gap between the first two. With uTCP, the application receives each segment as soon as it arrives, along with the sequence number information it needs to reconstruct a complete internal view of whichever fragments of the TCP stream have arrived.

The uTCP receiver retains in its receive buffer the TCP headers of segments received and delivered out-of-order, until its cumulative acknowledgment point moves past these segments, and generates acknowledgments and selective acknowledgments (SACKs) exactly as TCP normally would. The uTCP receiver does not increase its advertised receive window when it delivers data to the application out-of-order, so the advertised window tracks the cumulative in-order delivery point exactly as in TCP. In this fashion, uTCP maintains wire-visible behavior identical to TCP while delivering segments to the application out-of-order.

The uTCP receive path assumes the sender may be an unmodified TCP, and TCP's stream-oriented semantics allow the sending TCP to segment the sending application's stream at arbitrary points—independent of the boundaries of the sending application's `write()` calls, for example. Further, network middleboxes may silently *re-segment* TCP streams, making segment boundaries observed at the receiver differ from the sender's original transmissions [24]. An application using uTCP, therefore, must not assume anything about received segment boundaries. This is a key technical challenge to using uTCP reliably, and addressing this challenge is one function of uCOBS and uTLS, described later.

4.2 Sender-Side Enhancements

While uTCP's receiver-side enhancements address the "latency tax" on segments waiting in TCP's reordering buffer, TCP's sender-side queue can also introduce latency, as segments the application has already written to a TCP socket—and hence "committed" to the network—wait until TCP's flow and congestion control allow their transmission. Many applications can benefit from the ability to "late-bind" their decision on *what* to send until the last possible moment. An application-level multistreaming transport with prioritization, such as SST [19]

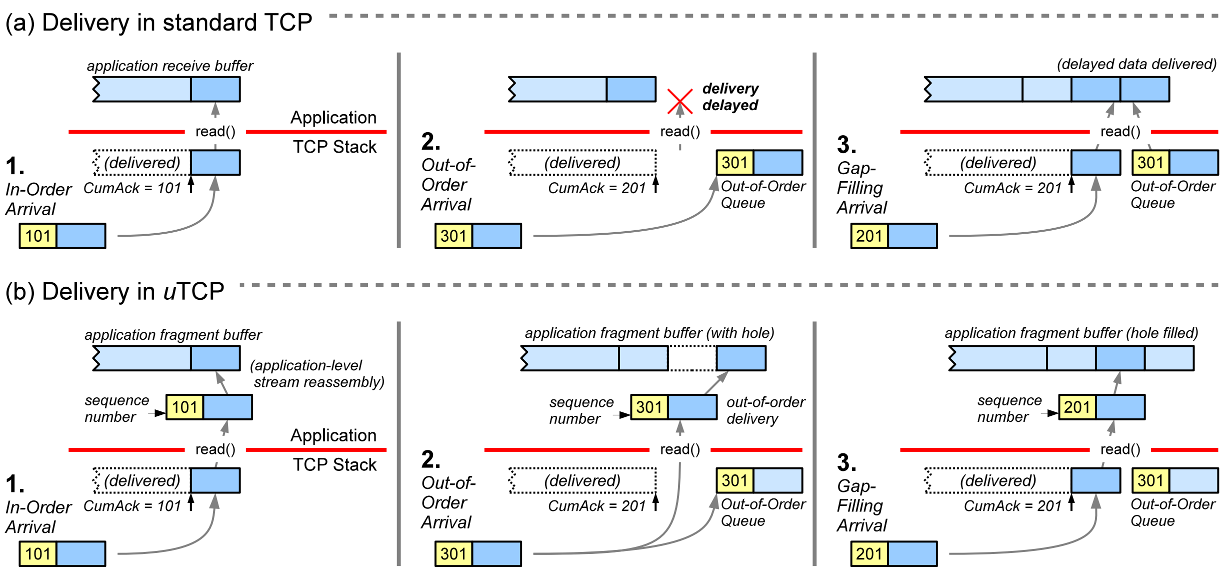


Figure 3: Delivery behavior of (a) standard TCP, and (b) uTCP, upon receipt of in-order and out-of-order segments.

or SPDY [2], would prefer high-priority packets not to get "stuck" behind low-priority packets in TCP's send queue. In applications such as games and remote-access protocols, where the receiver typically desires *only the freshest* in a stream of real-time status updates, the sender would prefer that new updates "squash" any prior updates still in TCP's send queue and not yet transmitted.

The Congestion Manager architecture [6] addressed this desire to "late-bind" the application's transmission decisions, by introducing an upcall-based API in which the OS performs no send buffering, but instead signals the application whenever the application is permitted to send. Upcalls represent a major change to conventional sockets APIs, however, and introduce issues such as how the OS should handle an application that fails to service an upcall promptly, leaving its allocated transmission time-slot unfilled yet unavailable to competing applications waiting to send.

In the spirit of maximizing deployability, uTCP adopts a more limited but less invasive design, by retaining TCP's send buffer but giving applications some control over it. After enabling uTCP's new socket option `SO_UNORDEREDSEND`, the OS expects any subsequent `write()` to that socket to include a short header, containing metadata that uTCP reads and strips before placing the remaining data on TCP's send buffer. The uTCP header contains an integer *tag* and a set of optional flags controlling uTCP's send-side behavior.

By default, uTCP interprets tags as priority levels. Instead of unconditionally placing the newly-written data at the tail of the send queue as TCP normally would, uTCP *inserts* the newly-written data into the send queue just *before* any lower-priority data in the send queue and not yet transmitted. The application thus avoids higher-priority packets being delayed by lower-priority packets enqueued earlier, while the OS avoids the complexity and security challenges of an upcall API.

For strict TCP wire-compatibility, uTCP never inserts new data into the send queue ahead of any previously-written data that has ever been transmitted in whole or in part: e.g., ahead of data from a prior application write already partly transmitted and awaiting acknowledgment. If an application writes a large low-priority buffer, then writes higher-priority data after transmission of the low-priority data has begun, uTCP inserts the high-priority data after the entire low-priority write and never in the middle. This constraint enables the application to control the boundaries on which send buffer reordering is permitted, independent of the current MTU and TCP segmentation behavior.

A simple uTCP refinement, which we intend to explore in future work, is to include a *squash* flag in the metadata header the application prepends to each write. If set, while inserting the newly-written data in tag-priority order, uTCP would also remove and discard any data previously written with exactly the same tag, that has not yet been transmitted in whole or in part. This refinement would enable update-oriented applications such as games to avoid the bandwidth cost transmitting old updates superseded by newer data.

4.3 Design Alternatives

uTCP pursues a conservative point in a large design space, and many alternatives present interesting tradeoffs. Some alternatives include: disabling TCP congestion control at the sender; assigning TCP sequence numbers at application write time instead of the time a segment is first transmitted; sending new data in retransmitted segments; modifying the receiver to acknowledge unreceived sequence space gaps for unreliable service; increasing the receive window to account for out-of-order segments; and delivering data to the application exactly-once instead of at-least-once. For space reasons we discuss these tradeoffs in more detail elsewhere [26]. A common theme, however, is that most of these design alternatives change TCP's behavior in wire-visible ways, which can trigger various unpredictable middlebox behaviors [24], making connectivity less reliable.

5 uCOBS: Simple Datagrams on TCP

Since uTCP's design attempts to minimize OS changes, its unordered delivery primitives do not directly offer applications a convenient, general-purpose datagram substrate. Minion's uCOBS protocol bridges this semantic gap, building atop uTCP (or standard TCP) a lightweight datagram delivery service comparable to UDP or DCCP. This first section first introduces the challenge of delimiting datagrams, then presents uCOBS' solution and discusses alternatives.

5.1 Self-Delimiting Datagrams for uTCP

Applications built on datagram substrates such as UDP generally assume the underlying layer preserves datagram boundaries. If the network fragments a large UDP datagram, the receiving host reassembles it before delivery to the application, and a correct UDP never merges multiple datagrams, or datagram fragments, into one delivery to the receiving application. TCP's stream-oriented semantics do not preserve any application-relevant frame boundaries within a stream, however. Both the TCP sender and network middleboxes can and do coalesce TCP segments or re-segment TCP streams in unpredictable ways [24]. Conventional TCP applications, which send and receive TCP data in-order, commonly address this issue by delimiting application-level frames with some length-value encoding, enabling the receiver to locate the next frame in the stream from the previous frame's position and header content.

Since uTCP's receive path effectively just bypasses TCP's reordering buffer, delivering received segments to the application as they arrive, a stream fragment received out-of-order from uTCP may begin at any byte offset in the stream, and not at a frame boundary meaningful to the application. Since the receiver is by definition missing some data sent prior to this out-of-order segment, it cannot rely on preceding stream content to compute the next frame's position.

Reliable use of uTCP, therefore, requires that frames embedded in the TCP stream be *self-delimiting*: recognizable without knowledge of preceding or following data. A simple solution is to make frames fixed-length, so the receiver can compute the start of the next frame from the stream offset uTCP provides with out-of-order segments. uCOBS is intended to offer a general-purpose datagram substrate, however, and many applications require support for variable-length frames.

If the application-level frames happen to be encoded so as never to include some "reserved" byte value, such as zero, then we could use that byte reserved value to delimit frames within uTCP streams. Since we wish uCOBS to support general-purpose delivery of datagrams of variable length containing arbitrary byte values, however, uCOBS must explicitly (re-)encode the application's datagrams in order to reserve some byte value to serve as a delimiter.

Any scheme that encodes arbitrary byte streams into strings utilizing fewer than 256 symbols will serve this purpose, such as the ubiquitous *base64* scheme, which encodes byte streams into strings utilizing only 64 ASCII symbols plus whitespace. Since base64 encodes three bytes into four ASCII symbols, however, it expands encoded streams by a factor of $4/3$, incurring a 33% bandwidth overhead. Since uCOBS needs to reserve only *one* byte value for delimiters, and not the large set of byte values considered "unsafe" in E-mail or other text-based message formats, base64 encoding is unnecessarily conservative for uCOBS' purposes.

5.2 Operation of uCOBS

To encode application datagrams efficiently, uCOBS employs *consistent-overhead byte stuffing*, or COBS [12]. COBS is analogous to base64, except that it encodes byte streams to reserve only *one* distinguished byte value (e.g., zero), and utilizes the remaining 255 byte values in the encoding. COBS could in effect be termed "base255" encoding. By reserving only one byte value, COBS incurs an expansion ratio of at most 255/254, or 0.4% bandwidth overhead.

Transmission: When an application sends a datagram, uCOBS first COBS-encodes the datagram to remove all zero bytes. uCOBS then prepends a zero byte to the encoded datagram, appends a second zero byte to the end, and writes the encoded and delimited datagram to the TCP socket. Since this sender-side encoding and transmission process operates entirely at application level within uCOBS, and does not rely on any OS-level extensions on the sending host, uCOBS operates even if the sender-side OS does not support uTCP.

The application can assign priorities to datagrams it submits to uCOBS, however, and if the sender's OS does support the uTCP extensions in Section 4.2, uCOBS passes these priorities to the uTCP sender, enabling higher-priority datagrams to pass lower-priority datagrams already enqueued. Since uTCP respects application `write()` boundaries while reordering the send queue, uCOBS preserves its delimiting invariant simply

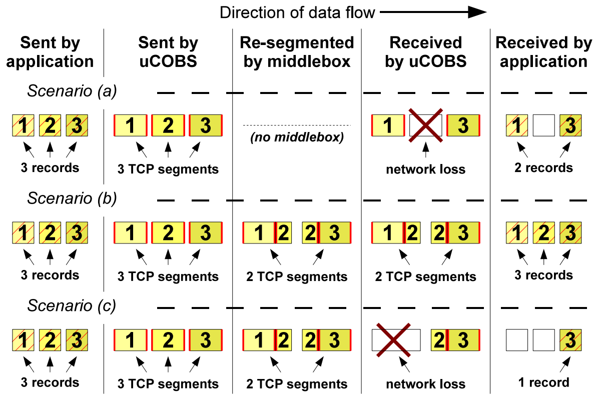


Figure 4: An example illustrating a uCOBS transfer

by writing each encoded datagram—with the leading and trailing zero bytes—in a single write.

Reception: At stream creation time, uCOBS enables uTCP's receive-side extensions if available. If the receive-side OS does not support uTCP, then uCOBS simply falls back on the standard TCP API, receiving, COBS-decoding, and delivering datagrams to the application in the order they appear in the TCP sequence space. (This may not be the application's original send order if the send-side OS supports uTCP.)

If the receive-side OS supports uTCP, then uCOBS receives segments from uTCP in whatever order they arrive, then fits them together using the metadata in uTCP's headers to form contiguous fragments of the TCP stream. The arrival of a TCP segment can cause uCOBS to create a new fragment, expand an existing fragment at the beginning or end, or "fill a hole" between two fragments and merge them into one. The portion of the TCP stream before the receiver's cumulative-acknowledgment point, containing no sequence holes, uCOBS treats as one large "fragment." uCOBS scans the content of any new, expanded, or merged fragment for properly delimited records not yet delivered to the application. uCOBS identifies a record by the presence of two marker bytes surrounding a contiguous sequence of bytes containing no markers or holes. Once uCOBS identifies a new record, it strips the delimiting markers, decodes the COBS-encoded content to obtain the original record data, and delivers the record to the application.

5.3 Why Two Markers Per Datagram?

For correctness alone, uCOBS need only prepend *or* append a marker byte to each record—not both—but such a design could reduce performance by eliminating opportunities for out-of-order delivery. Consider Scenario (a) in Figure 4, in which an application sends three records. uCOBS encodes these records and sends them via three `write()` calls, which TCP in turn sends in three separate TCP segments. In this scenario, no middleboxes re-segment the TCP stream in the network, but the middle segment is lost. If the uCOBS sender were only to *prepend* a marker at the start of each record, the uCOBS receiver could not deliver record 1 immediately on receipt, since it cannot tell if record 1 extends into the following "hole" in sequence space. Similarly, if the sender were only to *append* a marker at the end of each record, then uCOBS could not deliver segment 3 immediately on receipt, since record 3 might extend backwards into the preceding hole. By adding markers to both ends of each record, uCOBS ensures that the receiver can deliver each record as soon as all of its segments arrive.

These markers enable uCOBS to offer reliable out-of-order delivery even if network middleboxes re-segment the TCP stream. In Scenario (b) in Figure 4, for example, uCOBS sends three records encoded into three TCP segments as above, but a middlebox re-segments them into two longer TCP segments, whose boundary splits record 2 into two parts. If neither of these segments are lost, then the uCOBS receiver can deliver record 1 immediately upon receipt of the first TCP segment, and can deliver records 2 and 3 upon receipt of the second segment. If the first segment is lost as shown in Scenario (c), however, the uCOBS receiver cannot deliver the missing record 1 or the partial record 2, but can still deliver record 3 as soon as the second TCP segment arrives.

6 uTLS: Secure Datagrams on TCP

While uCOBS offers out-of-order delivery wire-compatible up to the TCP level, middleboxes often inspect and manipulate the *content* of TCP streams as well [38]. All unencrypted network traffic today is, *de facto*, "fair game" for middleboxes—and streams exhibiting any "out of the ordinary" middlebox-visible behavior are likely to fail over *some* middleboxes [24]. An application's only way to protect "end-to-end" communication in practice, therefore, is via end-to-end encryption and authentication. But network-layer mechanisms such as IPsec [27] face the same deployment challenges as new secure transports [19], and remain confined to the niche of corporate VPNs. Even VPNs are shifting from IPsec toward HTTPS tunnels [16], the only form of end-to-end encrypted connection almost universally supported on today's Internet. A network administrator or ISP might disable nearly any other port while claiming to offer "Internet access," but would be hard-pressed to disable HTTPS, today's foundation for E-commerce.

We could layer TLS atop uCOBS, but TLS decrypts and delivers data only in-order, negating uTCP's benefit. We could also layer the datagram-oriented DTLS [40] atop uCOBS, but the resulting (DTLS-encrypted *then* COBS-encoded) wire format would be radically different from TLS over TCP, and likely fail to traverse middleboxes expecting TLS, particularly on port 443.

The goal of uTLS, therefore, is to coax out-of-order delivery from the *existing* TCP-oriented TLS wire format, producing an encrypted datagram substrate indistinguishable on the wire from standard TLS connections (except via analysis of "side-channels" such as packet length and timing, which we do not address). Run on port 443, a uTLS stream is indistinguishable from HTTPS—regardless of whether the application actually uses HTTP headers, since the HTTP portion of HTTPS streams are TLS-encrypted anyway. Deployed this way, uTLS effectively offers an end-to-end protected substrate in the "HTTP as the new narrow waist" philosophy [36].

6.1 Design of uTLS

TLS [17] already breaks its communication into *records*, encrypts and authenticates each record, and prepends a header for transmission on the underlying TCP stream. TLS was designed to decrypt records strictly in-order, however, creating three challenges for uTLS:

- **Locating record headers out-of-order.** Since encrypted data may contain any byte sequence, there is no reliable way to differentiate a TLS header from record data in the TCP stream, as COBS encoding provides.
- **Encryption state chaining.** Some TLS ciphersuites chain encryption state across records, making records indecipherable until prior records are processed.
- **Record numbers used in MAC computation.** TLS includes a record number, which increases by 1 for each record, in computing the record's MAC. But the uTLS receiver may not know an out-of-order record's number: holes in TCP sequence space before the record could contain an unknown number of prior records.

To locate records out-of-order, uTLS first scans a received stream fragment for byte sequences that *may* represent the TLS 5-byte header: i.e., containing the correct record type and version, and a plausible length. While this scan may yield false positives, uTLS verifies the inferred header by attempting to decrypt and authenticate the record. If the cryptographic MAC check fails, instead of aborting the connection as TLS normally would, uTLS assumes a false positive and continues scanning.

Since TLS's MAC is designed to prevent resourceful adversaries from constructing a byte sequence the receiver could misinterpret as a record, and it is by definition at least as hard to find such a sequence "accidentally" as to forge one maliciously, TLS security should protect equally well against accidental false positives. One exception is when TLS is using its "null ciphersuite," which performs no packet authentication. With this ciphersuite, normally used only during initial key negotiation, uTLS disables out-of-order delivery to avoid the risk of accepting and delivering false records.

The only obvious solution to the second challenge above is to avoid ciphersuites that chain encryption state across records. Most ciphersuites before TLS 1.1 chain encryption state, unfortunately. Any stream cipher inherently does so, such as the RC4 cipher used in early SSL versions. Most recent ciphersuites use block ciphers in CBC mode. CBC ciphers do not inherently depend on chained encryption state, but do require an Initialization Vector (IV) for each record. Until recently, TLS produced each record's IV implicitly from the prior record's encryption state, making records interdependent.

To fix a security issue, however, TLS 1.1 block ciphers use explicit IVs, which the sender generates independently for each record and prepends to the record's ciphertext. As a side-effect, TLS 1.1 block ciphers support out-of-order decryption. Since TLS supports negotiation of versions and ciphersuites, uTLS simply leverages this process. An application can insist on TLS 1.1 with a block cipher to ensure out-of-order delivery support, or it can permit older ciphersuites to maximize interoperability, at the risk of sacrificing out-of-order delivery.

The third challenge is the implicit "pseudo-header" TLS uses in computing the MAC for each packet. This pseudo-header includes a "sequence number" that TLS increments once per *record*, rather than per *byte* as with TCP sequence numbers. When uTLS identifies a possible TLS record in a TCP fragment received out-of-order, the receiver knows only the byte-oriented TCP stream offset, and not the TLS record number. Since records are variable-length, unreceived holes prior to a record to be authenticated may "hide" a few large records or many smaller records, leaving the receiver uncertain of the correct record number for the MAC check.

To authenticate records out-of-order without modifying the TLS ciphersuite, therefore, uTLS attempts to *predict* the record's likely TLS record number, using heuristics such as the average size of past records, and may try several adjacent record numbers to find one for which the MAC check succeeds. If uTLS fails to find a correct TLS record number, it cannot deliver the record out-of-order, but will still eventually deliver the record in-order.

The current uTLS supports only receiver-side unordered delivery, and not the send-side uTCP enhancements in Section 4.2, because send-side reordering complicates record number prediction. A future enhancement we intend to explore is for uTLS to prepend an explicit record number to application payloads before encryption. Since encryption does not depend on record number, the receiving uTLS can decrypt the record number for use in the MAC check, avoiding the need to predict record numbers and enabling send-side reordering. Since the only wire-protocol change is protected by encryption, the change would be invisible to middleboxes. Preserving end-to-end backward compatibility may require a way to negotiate "under encryption" the use of explicit record numbers, however.

7 Prototype Implementation

This section describes the current Minion prototype, which implements uTCP in the Linux kernel, and implements uCOBS and uTLS in application-linked libraries. The uTCP prototype is Linux-specific, but we expect the API extensions it implements and the application-level libraries to be portable.

The uTCP Receiver in Linux: The uTCP prototype adds about 240 lines and modifies about 50 lines of code in the Linux 2.6.34 kernel, to support the new `SO_UNORDERED` socket option. This extension involved two main changes. First, uTCP modifies the TCP code that delivers segments to the application, to prepend a 5-byte metadata header to the data returned from each `read()` system call. This header consists of a 1-byte flags field and a 4-byte TCP sequence number. One flag bit is currently used, with which uTCP indicates whether it is delivering data in-order or out-of-order. Second, if TCP's in-order queue is empty, uTCP's `read()` path checks and returns data from the out-of-order queue. To minimize kernel changes, segments remain in the out-of-order queue after delivery, so uTCP will eventually deliver the same data again in-order.

The uTCP Sender in Linux: On the send path, uTCP adds about 250 lines of kernel code and modifies about 20 in Linux 2.6.34, supporting a new `SO_UNORDEREDSEND` socket option via two changes.

First, uTCP expects the application to prepend a 5-byte header, containing a 1-byte flags field and a 4-byte tag, to the data passed to each `write()`. The flags are currently unused, and the tag indicates message priority.

Second, uTCP inserts the data from each `write()` into the kernel's send queue in priority order. Linux's TCP send queue is a simple FIFO that packs application data into kernel buffers sized to the TCP connection's Maximum Segment Size (MSS). When inserting application messages non-sequentially, however, uTCP must preserve application message boundaries in the kernel. For simplicity, uTCP allocates kernel buffers (`skbuffs`) so that each message sent via uTCP starts a new `skbuff`, and may span several `skbuffs`, but no `skbuff` contains data from multiple application writes.

Disabling Linux's usual packing of MSS-sized `skbuffs` can affect Linux's congestion control, unfortunately, which counts `skbuffs` sent instead of bytes. Section 8.1 discusses the effects of this Linux-specific issue, which a future version of uTCP will address.

The uCOBS Library: The uCOBS prototype is a user-space library in C, amounting to ~700 lines of code [15]. uCOBS presents simple `cobs_sendmsg()` and `cobs_recvmsg()` interfaces enabling applications to send and receive COBS-encoded datagrams, taking advantage of send-side prioritization and out-of-order reception depending on the presence of send- and receive-side OS support for uTCP, respectively.

A uTLS Prototype Based on OpenSSL: The uTLS prototype builds on OpenSSL 1.0.0 [33], adding ~550 lines of code and modifying ~40 lines [15]. Applications use OpenSSL's normal API to create a TLS connection atop a TCP socket, then set a new uTLS-specific socket option to enable out-of-order, record-oriented delivery on the socket. OpenSSL 1.0.0 unfortunately does not yet support TLS 1.1, the first TLS version that uses explicit Initialization Vectors (IVs), permitting out-of-order decryption. For experimentation, therefore, the uTLS prototype modifies OpenSSL's TLS 1.0 ciphersuite to use explicit IVs as in TLS 1.1. Since this change breaks OpenSSL's interoperability, our prototype is not suitable for deployment. We are currently porting uTLS to the next major OpenSSL release, which supports TLS 1.1.

8 Performance Evaluation

This section evaluates Minion via experiments designed to approximate realistic application scenarios. All experiments were run across three Intel PCs running Linux 2.6.34: between two machines representing end hosts, a third machine interposes on the path and uses dummynet [9] to emulate various network conditions. To minimize well-known TCP delays fairly for both TCP and uTCP, we enabled Linux's "low latency" TCP code path via the `net.ipv4.tcp_low_latency` sysctl, and disabled the Nagle algorithm.

8.1 Bandwidth and CPU Costs

We first explore uTCP's costs, with and without record encoding and extraction via uCOBS and uTLS, for a 30MB bulk transfer on a path with 60ms RTT.

Raw uTCP: uTCP's CPU costs at both the sender and the receiver, without application-level processing, are almost identical to TCP's CPU costs, across a range of loss rates from 0% to 5% (figure omitted for space reasons).

Figure 5 shows bandwidth achieved by raw uTCP and TCP, for different application `write()` sizes. When the message size is a multiple of TCP's Maximum Segment Size (MSS)—at 1448 bytes (1 MSS) and 2896 bytes (2 x MSS)—uTCP's throughput is the same as TCP's.

The disparity elsewhere is due to Linux's congestion control counting `skbuffs` instead of bytes, mentioned earlier in Section 7. We partially address this issue by coalescing data into `skbuffs` where easily possible. More specifically, we coalesce small messages when they fully fit within MSS-sized `skbuffs` at the tail of the sender-side socket buffer. This fix makes uTCP throughput sim-

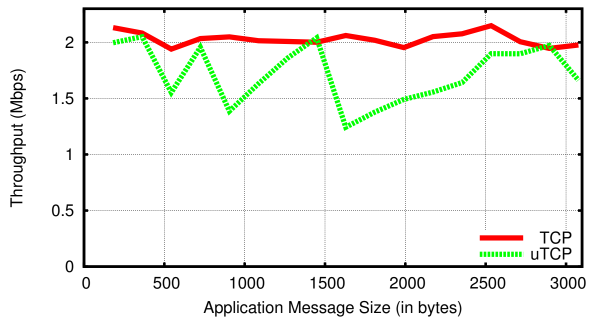


Figure 5: Throughput with different app message sizes.

ilar to TCP's when the MSS is divisible by message size—at 362 bytes ($\frac{1}{4}$ MSS) and 724 bytes ($\frac{1}{2}$ MSS). Future versions of uTCP will fully address this Linux-specific issue with changes either to uTCP or to Linux's congestion control.

Costs with uCOBS/uTLS: To measure these CPU costs, we run a 30MB bulk transfer over a path with a 60ms RTT, for several loss rates.

Figure 6(a) shows CPU costs including application-level encoding/decoding, atop standard TCP ("COBS") and atop uTCP ("uCOBS"), for several loss rates at both sender and receiver. The lighter part of each bar represents user time and the darker part represents kernel time. These results are normalized to the performance of raw TCP, with no application-level encoding or decoding.

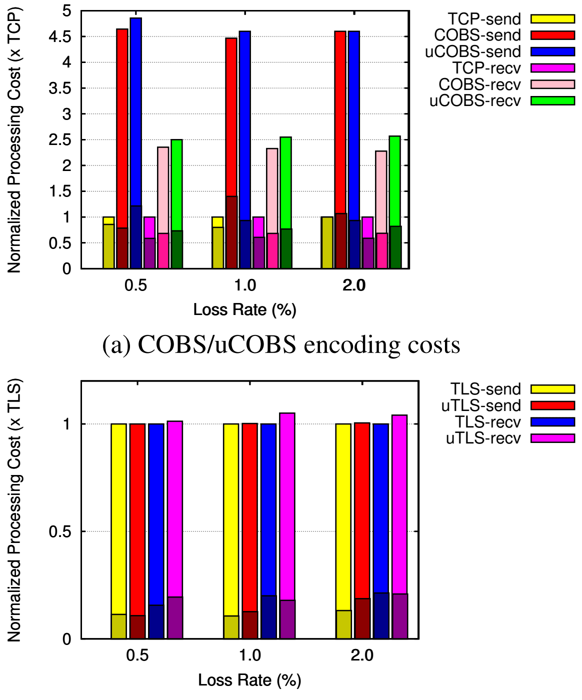
COBS encoding/decoding barely affects kernel CPU use but incurs some application-level CPU cost. This cost is partly due to the encoding itself, and partly because the libraries are not yet well-optimized.

Figure 6(b) shows the CPU costs of uTLS relative to TLS. At the sender, the CPU costs are identical, since there is nothing that uTLS does differently than TLS, and since the CPU cost of using uTCP is practically the same as with TCP. The user-space cost for the uTLS receiver is generally higher than TLS, since the uTLS receiver does more work in processing out-of-order frames than the TLS receiver, but this cost remains within 7% of the TLS receiver's cost.

The bandwidth penalty of uCOBS encoding is barely perceptible, under 1%. TLS's bandwidth overhead, up to 10%, is due to TLS headers, IVs, and MACs; uTLS adds no bandwidth overhead beyond standard TLS 1.1.

8.2 Conferencing Applications

We now examine a real-time Voice-over-IP (VoIP) scenario. A test application uses the SPEEX codec [50] to encode a WAV file using ultra-wideband mode (32kHz), for a 256kbps average bit-rate, and transmit voice frames at fixed 20ms intervals. Network bandwidth is 3Mbps and RTT is 60ms, realistic for a home broadband connection. To generate losses more realistically representing



(a) COBS/uCOBS encoding costs

(b) TLS/uTLS costs

Figure 6: CPU costs of using an application with TCP, COBS, and uCOBS at different loss rates.

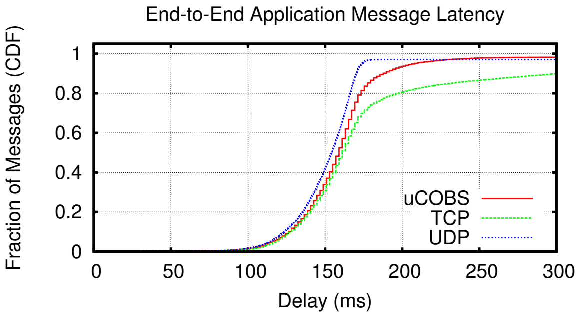


Figure 7: CDF of end-to-end latency in VoIP frames.

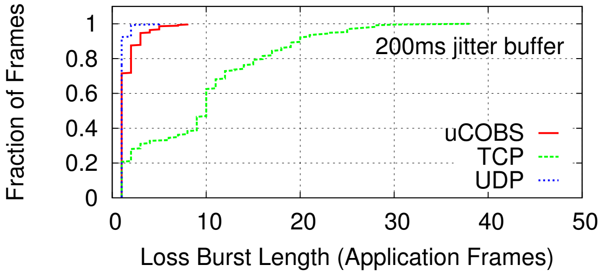


Figure 8: CDF of codec-perceived loss-burst size with TLV encoded frames over TCP, UDP, and uCOBS.

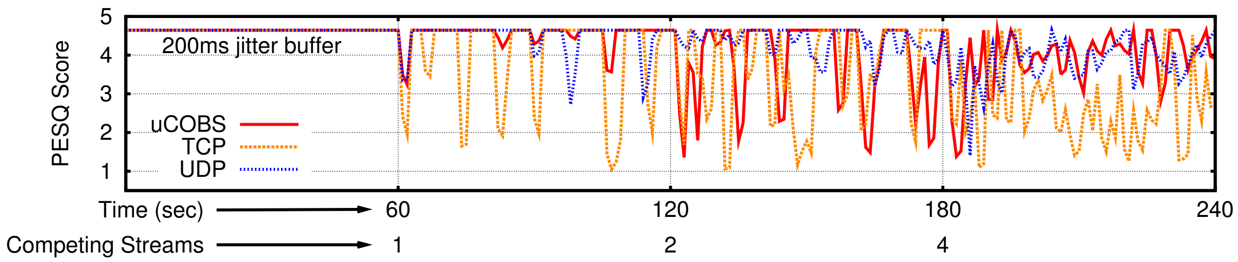


Figure 9: Moving PESQ score of VoIP call under increasing bandwidth competition.

network contention, we run a varying number of competing TCP file transfers, emulating concurrent web browsing sessions or a BitTorrent download, for example.

This is a simplistic scenario for experimental purposes. Real VoIP applications, which we intend to evaluate in future work, often determine bit-rate based on network conditions. Real applications may also implement loss recovery mechanisms atop UDP, which may improve perceived voice quality when using UDP.

Latency: Figure 7 shows a CDF of one-way per-frame latency perceived by the receiving application, under heavy contention from 4 competing TCP streams. All three transports suffer major delays. 4% of UDP frames do not arrive at all, since UDP does not retransmit. 95% of frames sent with uCOBS over uTCP arrive within 200ms, compared to 80% of TCP frames.

Burst Losses: VoIP codecs such as SPEEX can interpolate across one or two missing frames, but are sensitive to burst losses or delays, which yield user-perceptible blackouts. An application's susceptibility to blackouts depends on its jitter buffer size: a larger buffer increases the receiver's tolerance of burst losses or delays, but also increases effective round-trip delay, which can add user-perceptible "lag" to all interactions.

The CDF in Figure 8 shows the prevalence of different lengths of burst losses experienced by the receiver in a typical VoIP call. A burst loss is a series of consecutive voice frames that miss their designated playout time, due either to loss or delay.

A 200ms jitter buffer of $3\times$ the path RTT might seem generous, but the ITU's recommended maximum transmission time of 400ms [4] allows for a larger buffer with these network conditions. Now the differences between uCOBS and TCP are quite pronounced, with 80% of burst losses atop uCOBS being 3 or fewer packets, nearly matching that of UDP. Meanwhile 40% of TCP's bursts are greater than 10 packets, producing highly-perceptible $1/5$-second pauses.

Perceptual Audio Quality: To illustrate the impact of unordered delivery on VoIP quality, we use Perceptual Evaluation of Speech Quality (PESQ) [32] to measure audio reproduction quality, by comparing the audio stream reproduced by SPEEX at the receiver against that of an ideal run with no lost or delayed frames. We transmit a 4-minute VoIP call using a jitter buffer of 200ms, introducing 1 to 4 competing TCP streams progressively at 1-minute intervals.

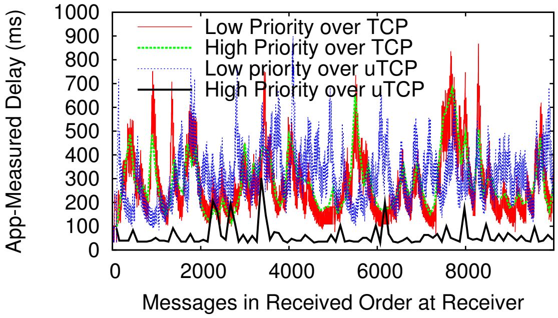


Figure 10: Prioritized messages experience lower end-to-end delay with uTCP.

Figure 9 plots PESQ quality scores for 2-second sliding time windows over a representative 4-minute call, comparing transmission via uCOBS, TCP, and UDP. The effect of network contention becomes apparent even with only one competing stream, but unordered delivery makes this impact much smaller on uCOBS or UDP than on TCP. uCOBS sometimes performs better than UDP, in fact, when uTCP successfully retransmits a lost segment within the jitter buffer's time window, whereas UDP never retransmits. (Some UDP applications employ application-level retransmission schemes [1], especially for control data.) Like TCP, uCOBS shows greater volatility than UDP with higher contention, due to TCP congestion control effects that uTCP preserves (though congestion control can be disabled). Similarly, the "back-off" of the *competing* streams enables the transports to rebound after the initial contention of 4 competing streams.

8.3 Send-Side Prioritization

To test uTCP's sender-side prioritization, we use a synthetic application that continuously sends messages to the receiver at network-limited rate, of which one in every 100 messages are considered "high-priority." Fig-

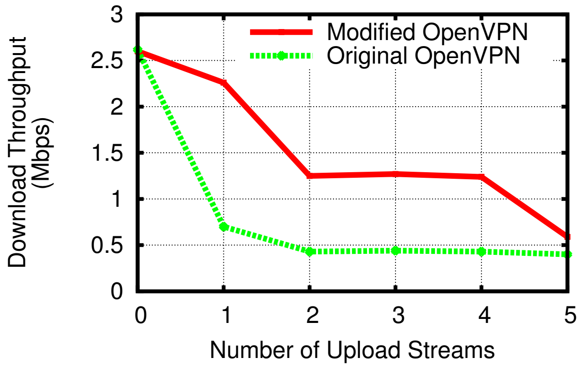


Figure 11: Throughput obtained by a TCP flow through modified and unmodified OpenVPN.

ure 10 plots application-observed messages delay over time, for high- and low-priority messages, atop TCP versus uTCP. As expected, high-priority messages consistently observe much lower delays under uTCP because they short-cut the TCP send queue. The next section explores a more realistic application for prioritization.

8.4 VPN Tunneling

Applications running atop TCP-based VPN tunnels often encounter *TCP-in-TCP* effects [48]. The applications' tunneled TCP flows assume they are running atop a best-effort, packet-switched network as usual, but are in fact running atop a reliable, in-order TCP-based tunnel. The TCP tunnel affects the tunneled flows' congestion control by increasing observed latency and RTT variance, and masks losses: tunneled flows never see "lost" or "reordered" TCP segments, only long-delayed ones. While uTCP does not change TCP's reliability or congestion control, it offers tunneled flows lower delay and jitter, and a more accurate view of packet losses.

To test Minion's impact on TCP-in-TCP effects, we made two changes to OpenVPN 2.1.4 [34]. First, we modified OpenVPN to use uCOBS instead of TCP, enabling unordered delivery of tunneled IP packets. Second, to reduce delay variance of tunneled TCP flows further, the modified OpenVPN gives tunneled TCP ACKs a higher priority than other packets.

The experiment uses a link with 3Mbps download and 0.5Mbps upload bandwidth, consistent with the median speed of residential connections [53].

Figure 11 shows measured throughput of a single *download*, with original and modified OpenVPN tunnels, for a varying number of competing *uploads* within the same tunnel. While using uTCP does not eliminate all TCP-in-TCP effects, the reduced RTT and RTT-variance noticeably improve performance.

To understand these performance improvements further, we now measure total network utilization achieved

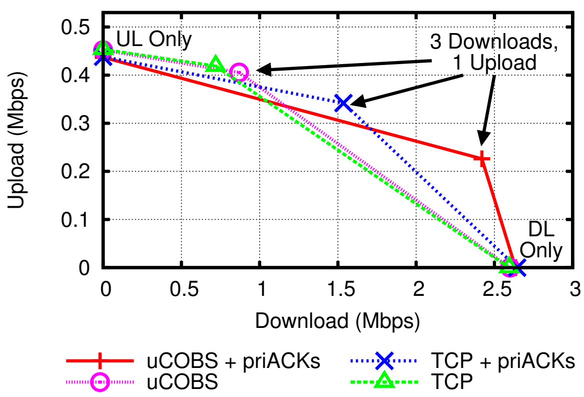


Figure 12: Contribution of independent modifications to network utilization.

by independently adding the two modifications—unordered delivery at the receiving ends of the tunnel (labeled *uCOBS*), and ACK prioritization at the sending ends (labeled *priACKs*)—leading to four variants of OpenVPN. Figure 12 shows total upload and download throughputs obtained by the VPN tunnel in three different contexts: one upload (labeled *UL*) within the tunnel, one upload competing with three downloads within the tunnel, and one download (labeled *DL*) in the tunnel.

With no competing flows, labeled *UL Only* and *DL Only* in Figure 12, all four variants perform similarly. With multiple competing downloads, out-of-order delivery improves download performance by a small amount, but ACK prioritization greatly improves download performance. Upload throughput suffers, however, as ACK prioritization is added. This throughput degradation is atrributable to the poor interaction between the small `write()`s of ACK packets being sent through the tunnel and Linux's `skbuff`-based congestion control described in Section 8.1. Despite this degradation to upload throughput, the area under the curve—representing total network utilization—remains highest with the fully modified tunnel. In a future version of uTCP that fixes this Linux-specific issue, we expect the upload throughput to remain high even with ack-prioritization, and network utilization to reflect uTCP's benefits more clearly.

8.5 Multistreaming Web Transfers

To explore uTCP's potential benefits for web browsing, we built msTCP, a simple *multistreaming* protocol providing multiple concurrent, ordered message streams atop a single uTCP connection. While similar in purpose to SPDY [2], to our knowledge msTCP is the first TCP-based multistreaming protocol that offers the unordered delivery benefits of non-TCP-based multistreaming protocols such as SCTP [45] and SST [19]. We omit a detailed description of msTCP for space reasons, but its design follows standard techniques.

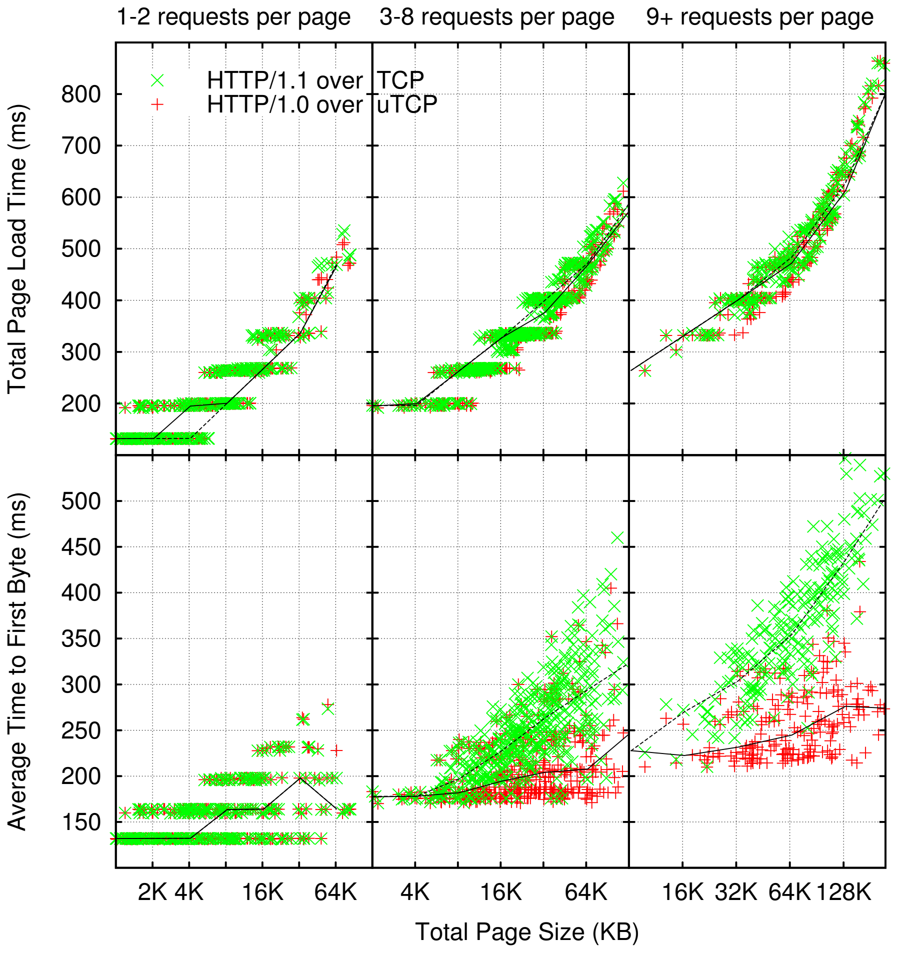


Figure 13: Pipelined HTTP/1.1 over a persistent TCP connection, vs. Parallel HTTP/1.0 over msTCP.

To evaluate msTCP, we compare the performance of parallel HTTP/1.0-style object requests over msTCP, against pipelined HTTP/1.1 requests on a persistent TCP connection, under a trace-driven web workload. We use a fragment of the UC Berkeley Home IP web client traces from the Internet Traffic Archive [25], using the trace to drive a series of web page downloads. Each page consists of a "primary" request for the HTML, followed by "secondary" requests for embedded objects such as images. The simulation pessimistically assumes that the browser cannot begin requesting secondary objects until it has downloaded the primary object completely, but at this point it can request all secondary objects in parallel. The experimental setup uses a link with 1.5Mbps bandwidth in each direction and with a 60ms RTT.

Figure 13 shows a scatter-plot of total page load time in the top three graphs, and in the bottom three graphs, average time to load the first byte of an object in each page—the time at which the browser can potentially start rendering the object. The dark curves show median times, computed across buckets of web page sizes. As the figure shows, msTCP does not affect total page load times noticeably. msTCP shows much lower delay in *starting* to load many objects, however, since msTCP interleaves different objects' chunks within the persistent connection.

Figure 13 shows the end-to-end impact on web browsing of msTCP's application-level message chunking and multiplexing in addition to the benefits of uTCP's out-of-order delivery. These latency savings, while not solely due to uTCP, represent the potential savings when web frameworks like SPDY [2] use uTCP, and make HTTP/HTTPS more usable as a general purpose substrate for deploying latency-sensitive applications [36].

8.6 Implementation Complexity

To evaluate the implementation complexity of uTCP and the related application-level code, Table 1 summarizes the source code changes uTCP makes to Linux's TCP stack in lines of code [15], the size of the standalone uCOBS library, and the changes uTLS makes to OpenSSL's `libssl` library. The SSL/TLS total does not

	TCP	uTCP	DCCP	SCTP
Kernel Code	12,982	**565** (4.6%)	6,338	19,312

	uCOBS	SSL/TLS	uTLS	DTLS
User Code	**732**	31,359	**586** (1.9%)	4,734

Table 1: Code size of uTCP prototype as a delta to Linux's TCP stack, the uCOBS library, and uTLS as a delta to `libssl` from OpenSSL. Code sizes of "native" out-of-order transports are included for comparison.

include OpenSSL's `libcrypt` library, which `libssl` requires but uTLS does not modify.

With only a 600-line change to the Linux kernel and less than 1400 lines of user-space support code, uTCP provides a delivery service comparable to Linux's $6,300$-line native DCCP stack, while providing greater network compatibility. In user space, uTLS represents less than a 600-line change to the stream-oriented SSL/TLS protocol, contrasting with OpenSSL's $4,700$-line implementation of DTLS, which runs only atop out-of-order transports such as UDP or DCCP.

9 Related Work

New transports for latency-sensitive apps: Brosh et al. [8] model TCP latency, and identify the regions of operation for latency-sensitive apps with TCP. While some of the considerations apply, such as latency induced by TCP congestion control, uTCP extends the working region for such apps by eliminating delays at the receiver.

DCCP [28, 29] provides an unreliable, unordered datagram service with negotiable congestion control. SCTP [45] provides unordered and partially-ordered delivery services to the application. Both DCCP and SCTP face large deployment barriers on today's Internet, however, and are thus not widely used.

New transports such as SST [19] and CUSP [47] run atop UDP to increase deployability, and UDP tunneling schemes have been proposed for standardized Internet transports as well [35, 49]. Many Internet paths block UDP traffic as well, however, as evidenced by the shift of popular VoIP applications such as Skype [7] and VPNs such as DirectAccess [16] toward tunneling atop TCP instead of UDP, despite the performance disadvantages.

Message Framing over TCP: Protocols such as HTTP [18], SIP [42], and iSCSI [43], can all benefit from out-of-order delivery, but use TCP for legacy and network compatibility reasons. All use simple type-length-value (TLV) encodings, which do not directly support out-of-order delivery even with uTCP, because they offer no reliable way to distinguish a record header from data in a TCP stream fragment. While COBS [12] represents an attractive set of characteristics for framing records to enable out-of-order delivery, other encodings such as BABS [10] also represent viable alternatives.

10 Conclusion

For better or worse, TCP remains the most common substrate for application-level protocols and frameworks, many of which can benefit from unordered delivery. Minion demonstrates that it is possible to obtain unordered delivery from wire-compatible TCP and TLS streams with surprisingly small changes to TCP stacks and application-level code. Without discounting the value of UDP and newer OS-level transports, Minion offers a more conservative path toward the performance benefits of unordered delivery, which we expect to be useful to applications that use TCP for a variety of pragmatic reasons.

Acknowledgments

We wish to thank Jeff Wise, Dishant Ailawadi, Stuart Cheshire, Matt Mathis, Kun Tan, and the anonymous reviewers for their valuable feedback and contributions to early drafts. This research was sponsored by the NSF under grants CNS-0916413 and CNS-0916678.

References

[1] OpenArena project. `http://openarena.ws/`.

[2] SPDY: An Experimental Protocol For a Faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[3] ZeroMQ: The intelligent transport layer. `http://www.zeromq.org`.

[4] ITU. Recommendation G.114: One-way transmission time, May 2003.

[5] F. Audet, ed. and C. Jennings. Network address translation (NAT) behavioral requirements for unicast UDP, Jan. 2007. RFC 4787.

[6] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *SIGCOMM*, Sept. 1999.

[7] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *INFOCOM*, Apr. 2006.

[8] E. Brosh, S. A. Baset, V. Misra, D. Rubenstein, and H. Schulzrinne. The delay-friendliness of TCP for real-time traffic. *IEEE Transactions on Networking*, 18(5):1478–1491, 2010.

[9] M. Carbone and L. Rizzo. Dummynet Revisited. *ACM CCR*, 40(2), Apr. 2010.

[10] J. S. Cardoso. Bandwidth-efficient byte stuffing. In *IEEE ICC 2007*, 2007.

[11] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues, Feb. 2002. RFC 3234.

[12] S. Cheshire and M. Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, Sept. 1997.

[13] Cisco. Rate-Based Satellite Control Protocol, 2006.

[14] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[15] A. Danial. Counting Lines of Code, ver. 1.53. http://cloc.sourceforge.net/.

[16] J. Davies. DirectAccess and the thin edge network. *Microsoft TechNet Magazine*, May 2009.

[17] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.

[18] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.

[19] B. Ford. Structured streams: a new transport abstraction. In *SIGCOMM*, Aug. 2007.

[20] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets-VII*, Oct. 2008.

[21] B. Ford and J. Iyengar. Efficient cross-layer negotiation. In *HotNets-VIII*, Oct. 2009.

[22] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT behavioral requirements for TCP, Oct. 2008. RFC 5382.

[23] L. Guo, E. Tan, S. Chen, Z. Xiao, O. Spatscheck, and X. Zhang. Delving into Internet Streaming Media Delivery: a Quality and Resource Utilization Perspective. In *IMC*, Oct. 2006.

[24] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *IMC*, Nov. 2011.

[25] The Internet traffic archive. http://ita.ee.lbl.gov/.

[26] J. Iyengar, S. O. Amin, M. F. Nowlan, N. Tiwari, and B. Ford. Minion: Unordered delivery wire-compatible with TCP and TLS (full version), Apr. 2012. Technical Report. arXiv:1103.0463.

[27] S. Kent and K. Seo. Security architecture for the Internet protocol, Dec. 2005. RFC 4301.

[28] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), Mar. 2006. RFC 4340.

[29] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *SIGCOMM*, 2006.

[30] J. Mogul. TCP offload is a dumb idea whose time has come. In *HotOS IX*, May 2003.

[31] P. Natarajan et al. SCTP: An innovative transport layer protocol for the Web. In *15th WWW*, May 2006.

[32] I.-T. T. S. S. of ITU. Wideband extension to recommendation p.862 for the assessment of wideband telephone networks and speech codecs, Nov. 2007.

[33] OpenSSL project. http://www.openssl.org/.

[34] OpenVPN project. http://openvpn.net/.

[35] T. Phelan. DCCP Encapsulation in UDP for NAT Traversal (DCCP-UDP), Aug. 2010. Internet-Draft draft-ietf-dccp-udpencap-02 (Work in Progress).

[36] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *HotNets-IX*, Oct. 2010.

[37] J. Postel. User datagram protocol, Aug. 1980. RFC 768.

[38] C. Reis et al. Detecting in-flight page changes with web tripwires. In *5th NSDI*, Apr. 2008.

[39] E. Rescorla. HTTP over TLS, May 2000. RFC 2818.

[40] E. Rescorla and N. Modadugu. Datagram transport layer security, Apr. 2006. RFC 4347.

[41] J. Rosenberg. UDP and TCP as the new waist of the Internet hourglass, Feb. 2008. Internet-Draft (Work in Progress).

[42] J. Rosenberg et al. SIP: session initiation protocol, June 2002. RFC 3261.

[43] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI), Apr. 2004. RFC 3720.

[44] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, July 2003. RFC 3550.

[45] R. Stewart, ed. Stream control transmission protocol, Sept. 2007. RFC 4960.

[46] Transmission control protocol, Sept. 1981. RFC 793.

[47] W. W. Terpstra, C. Leng, M. Lehn, and A. P. Buchmann. Channel-based unidirectional stream protocol (CUSP). In *INFOCOM Mini Conference*, Mar. 2010.

[48] O. Titz. Why TCP over TCP is a bad idea, Apr. 2001. http://sites.inka.de/bigred/devel/tcp-tcp.html.

[49] M. Tuexen and R. Stewart. UDP Encapsulation of SCTP Packets, Jan. 2010. Internet-Draft draft-tuexen-sctp-udp-encaps-05 (Work in Progress).

[50] J.-M. Valin. The speex codec manual version 1.2 beta 3, Dec. 2007. http://www.speex.org/.

[51] D. Velten, R. Hinden, and J. Sax. Reliable data protocol, July 1984. RFC 908.

[52] W3C. The websocket api (draft), 2011. http://dev.w3.org/html5/websockets/.

[53] www.speedmatters.org. 2010 report on internet speeds in all 50 states, Nov. 2010. http://www.speedmatters.org/content/internet-speed-report.

[54] M. Zec, M. Mikuc, and M. Zagar. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. In *SoftCOM*, 2002.