Unordered Delivery in TLS-Encrypted TCP Connections

690 Report
Department of Computer Science
Yale University

Michael F. Nowlan

Advisor: Bryan Ford

ABSTRACT

TCP and UDP offer markedly different transport semantics. However, increasingly, applications robust to the unreliability of UDP choose TCP because it is more likely to successfully navigate today's Internet full of meddlesome middleboxes (ie. firewalls and NATs). The Transport Next Generation (Tng) project attempts to alleviate the logjam caused by this shifting of the Internet's narrow-waist towards TCP. In this paper, we present Unordered TLS (uTLS), the second of Tng's two new proposed transport modes. uTLS and Unordered TCP (uTCP) together enable out-of-order delivery of application data while maintaining strict wire-compatibility with TLS and TCP. We explore the requirements and implementation of uTLS, and its ability to "go between" UDP and TCP by emulating both stream and datagram behavior. We achieve less than a 3.7% performance increase compared to standard TLS with an even smaller increase in code. With promising early results, we discuss how further development can demonstrate the effectiveness of uTLS in real-time applications, such as Voice-Over IP.

1. INTRODUCTION

TCP's reliable, in-order delivery service [22], designed for application convenience, comes at a fundamental cost of delaying data delivery to the application. When the network loses one data segment, the receiving TCP must buffer and delay all segments within at least the next round-trip time (RTT), until the sender reacts and successfully retransmits the lost segment. Many applications, such as audio/video conferencing and VPN tunneling, tolerate one packet's outright loss more gracefully than the delay of a full RTT worth of packets, making these applications ill-suited to TCP.

Recognizing the needs of delay-sensitive applications, all standardized transports since TCP [14, 17, 21, 26], and various experimental transports [10, 23], offer out-of-order delivery. Yet factors such as TCP's inertia, and the proliferation of firewalls and NATs, have impeded the deployment of new transports [11, 16, 20]. As a result, modern delay-sensitive applications, such as the Skype telephony system [2] and Microsoft's DirectAccess VPN [4], regularly use TCP despite its performance drawbacks, in order to maximize their chance of functioning at all over adverse network paths.

The Transport Next Generation (Tng) project aims to alleviate the tension in choosing between reachability with delay and out-of-order data delivery. To do this, we observe that it is usually not the *network* but rather the *receiving TCP stack* that withholds out-of-order segments from the application, introducing TCP's delivery delays. To work around this, the Tng project adds an extension to TCP's API that enables out-of-order delivery within TCP and TLS. These modified versions of the protocols are identical on-the-wire as the originals, but simply enable applications to request out-of-order data. Unordered TCP (uTCP) and Unordered TLS (uTLS) merely expose information to the application that TCP stacks traditionally hide.

In this paper, we explore the design and implementation of uTLS, and how it complements uTCP to fulfill the larger goals of the Tng project.

As deep packet inspection middleboxes have proliferated, unfortunately, only TCP streams containing HTTP [9] or HTTPS [18] (TLS-over-TCP) now traverse many paths reliably [16]. To surmount this further compatibility challenge, we offer uTLS, a version of TLS [7] modified to support out-of-order record delivery. uTLS does not modify the TLS wire protocol seen by the network. Instead, the uTLS receiver scans TCP stream fragments for TLS records in their standard encoding, then authenticates, decrypts, and delivers them to the application *out-of-order*. Achieving this compatibility presents additional challenges—false positives in the scanning process, cryptographic interdependencies between records, and the record numbers TLS uses in MAC verification—but uTLS works around these challenges.

Experiments testing the performance of uTCP and uTLS show that these approaches successfully offer the delay benefits of out-of-order delivery on typical Internet paths.

This paper's primary contributions are: (a) a modification to TLS that enables out-of-order record delivery without modifying the standard TLS wire format; and (b) experimental evidence that a uTLS prototype implementation incurs minor overhead in terms of code and resources.

2. MOTIVATION

The design of the Internet encourages growth by minimizing the requirements for communication between hosts. The Internet Protocol (IP) specifies how to identify and reach hosts uniquely on the Internet. Below this layer, the physical implementation varies, with options such as wireless, ethernet and satellite. Above IP, there are multiple transports such as TCP, UDP and SCTP. Each of these physical and transport layer options has its own benefits and drawbacks. This tradeoff enables the Internet to support rich and diverse applications that demand different semantics. With IP as the "narrow waist", or only requirement, of network applications, this fosters innovation and growth by allowing applications to define functionality and behavior as needed.

2.1 Shifting of the Narrow Waist

With the evolution of the Internet and web browsing, TCP's reliable and in-order data delivery semantics likely gave it an advantage over other transports, such as UDP. With the World Wide Web and its use of HTTP, much Internet traffic flows over the HTTP-on-TCP-on-IP layering.

The increased importance of the Internet and its applications to society have forced network operators to support as much Internet traffic as possible. Given that so many applications use TCP-on-IP, an easy way to begin supporting the end-user is to ensure the network handles TCP traffic correctly. Supporting other transports is a distant second priority. As a result, non-TCP transports are susceptible to unreachability in some network paths.

A second consequence of the penetration and growing importance of the Internet is security. The potential devastation resulting from network vulnerabilities prompts network operators to block any communication that seems suspicious. Given the popularity of TCP described above, it is easy to understand that "suspicious" really means "anything but TCP". Firewalls and Network Address Translators (NATs) by default may block traffic using other transports, funneling Internet applications into the well-known and understood behavior of TCP-on-IP, or HTTP-on-TCP-on-IP.

As a result of its familiarity and widespread support, TCP remains the only transport that ensures connectivity.

2.2 TCP/TLS Tunnels

This shifting towards TCP-on-IP and TLS-on-TCP-on-IP has consequences for applications that wish to use other transports. Rather than using other transports directly on IP, applications use TCP tunnels to communicate, encapsulating application semantics or eliminating them altogether.

Emerging studies in the industry show that the shifting of the narrow waist has real-world impact.

- **Media Streaming/Conferencing:** Real-time applications such as VoIP and media streaming, which traditionally used UDP for transport, increasingly use TCP instead. Most commercial media streaming traffic now flows atop TCP—over 70% in a recent study [12]. While video-on-demand services can smooth over TCP's artificial delays using jitter buffers a few seconds long, "face-to-face" VoIP and videoconferencing applications have no such luxury since long round-trip delays are perceptible and frustrating to users. Nevertheless, teleconferencing applications such as Skype often choose TCP over UDP [2].
- **New Transport Services:** Recognizing that evolutionary developments have moved the *de facto* "narrow waist" of the Internet upward to include at least TCP and perhaps even HTTP [11, 16, 20], new transport services increasingly choose to tunnel atop TCP or HTTP to avoid being blocked by middleboxes. Recent examples include the W3C's WebSocket API [27] and Google's SPDY [1].
- **Virtual Private Networks (VPNs):** To provide reliable remote access to enterprise environments, VPNs are increasingly moving from "raw" IPSEC tunnels [13] toward TLS-over-TCP tunnels, as in Microsoft's DirectAccess [4]. Since both the tunnel itself and the tunneled traffic often use TCP, these VPNs produce deep recursive layer cakes, e.g., "TCP-on-IPv6-on-HTTP-on-TLS-on-TCP-on-IPv4," often yielding unexpected performance side-effects [24].

3. TRANSPORT NEXT GENERATION

The Transport Next Generation (Tng) project attempts to alleviate some of the pressure caused when applications are forced to use TCP, though they desire more UDP-like semantics. The Tng project modifies TCP and TLS to directly support a datagram delivery service to the application. Additionally, Tng proposes breaking the Transport Layer into multiple layers, each handling a specific function such as naming, negotiation and flow regulation.

3.1 Unordered TCP (uTCP)

To coax out-of-order data from the receiving TCP stack, we've added a kernel modification to the Linux kernel that enables an application to request an "unordered" socket option. With this option enabled, the kernel still delivers contiguous data. However, the first byte delivered may be many bytes beyond the cumulative ACK point, if there is a gap of data that has not yet been received. Our semantics ensure that all application data is delivered in order *at least* once. With this modification, an application will receive data that would otherwise be delayed for a lost TCP segment. Furthermore, because the underlying transport is still TCP, the application is guaranteed to receive every byte at least once, due to TCP's reliability.

We call this kernel modification Unordered TCP, or uTCP. Figure 1 shows the architecture of the receiving TCP Stack with our kernel modification.

3.1.1 Datagram Semantics and Offering More

To fully emulate datagram delivery, as in UDP, uTCP needs to encode message boundaries within the data. Applications using UDP expect message boundaries to be encoded at the

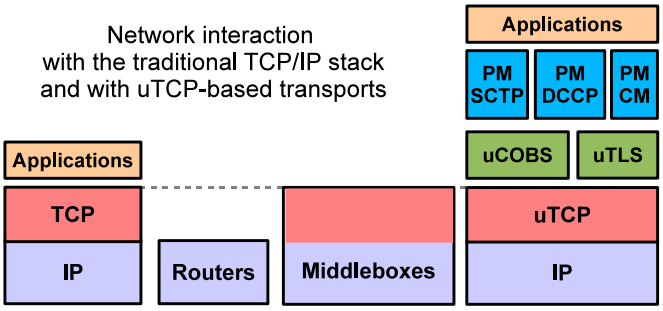

Figure 1: Proposed Tng architecture, including uTLS, uCOBS and uTCP, versus traditional TCP/IP.

time of writing, preserved through transmission in the network, and enforced on delivery to the receiving application. In contrast, TCP's stream-oriented semantics allow for network re-segmentation of the data because the receiving host will always deliver in-order. Because the network may re-segment data during transmission, the receiver can not guarantee that reads occur on the same boundaries as writes.

Applications using datagram semantics will likely use their own message sequence numbers, if necessary. Although we could abstract away the organization of such information into uTCP, we take the minimalist approach, and offer no more than what UDP semantics state. As a result, we must offer an encoding scheme that preserves message boundaries within the TCP stream so that the receiving stack can know when it has a complete message.

3.1.2 COBS encoding

Any encoding scheme may be used to mark boundaries, and we choose Consistent Overhead Byte Stuffing (COBS) [3]. COBS works by removing a special delimeter byte from a data stream and using this delimeter to mark the message boundaries. By bookending each message with the delimeter, the receiver can scan a contiguous byte stream and detect the presence of complete messages. When used in conjunction with uTCP, uCOBS offers full datagram delivery emulation to the application while maintaining TCP wire format.

3.2 uTCP in Action

Figure 2 illustrates the behavior of uTCP compared to TCP as viewed by the application in a simple bulk transfer experiment. We use this experiment solely to demonstrate the advantage of uTCP over TCP: that the application continues to receive new data despite dropped segments.

In summary, the Tng project aims to jumpstart Internet growth in the transport layer. It does this by making all transports wire-compatible with TCP and TLS. By ferrying traffic over these two protocols, the chance of a failed connection decreases. In order to support all transports within TCP and TLS, we proposed a kernel modification for out-of-order delivery. An application wishing to use out-of-order data delivery has two choices: it can use the uCOBS for a full datagram delivery emulation, or it can use the underlying socket directly (ie. only uTCP), and order the data itself. uTLS, the focus of this paper, uses the second approach.

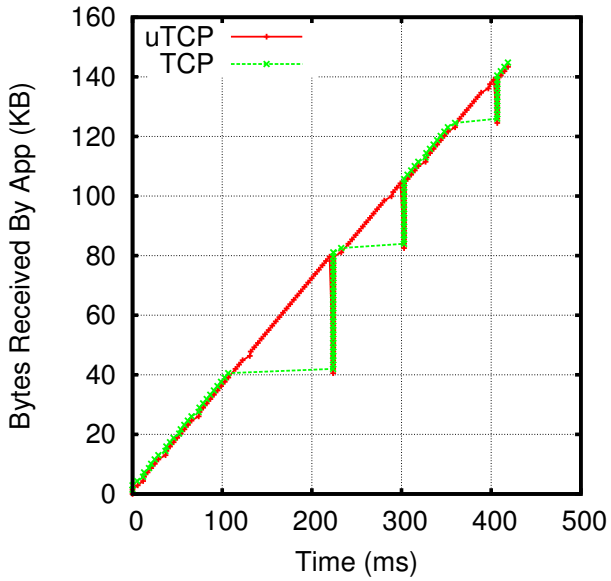

Figure 2: uTCP delivers subsequent data after a lost packet, rather than delaying it as in TCP.

4. GOING UNORDERED IN TLS

While uCOBS offers out-of-order record delivery wire-compatible up to the TCP level, middleboxes often inspect the *content* of TCP streams as well, via Deep Packet Inspection and even application payload manipulation. An increasingly *de facto* rule is that anything *not* encrypted in a TCP or UDP stream is "fair game" for middleboxes. An application's only way to ensure "end-to-end" communication in practice, therefore, is via end-to-end encryption and authentication. But network-layer mechanisms such as IPsec [13] face the same deployment challenges as new secure transports [10], and remain confined to the niche of corporate VPNs. Even VPNs are shifting from IPsec toward HTTPS tunnels [4], 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 SSL/TLS connections to port 443, the Web's foundation for now-crucial E-Commerce.

We could layer TLS directly atop uCOBS, but TLS normally decrypts and delivers data only in-order and thus would eliminate uTCP's benefit. More appropriate would be layering DTLS [19], the datagram-oriented version of TLS, atop uCOBS, but the resulting COBS-encoded, DTLS-encrypted records would have a wire format radically different from TLS over TCP. This new encoding would be unfamiliar and likely suspicious to middleboxes, and may be unusable on the crucial 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 out-of-order delivery channel essentially indistinguishable from standard TLS connections (other than via "side-channels" such as packet length and timing, which we do not address here).

4.1 Typical TLS

TLS [7] already breaks its communication into *records*, encrypts and authenticates each record, and prepends a version/type/length header for transmission on the underlying TCP stream. Of these steps, encryption and authentication require specific inputs. Encryption requires the data, cryptographic key and 16-byte Initialization Vector (IV) in order to produce cipher text. This cipher text is then hashed with a sequence, or record, number producing a Message Authentication Check (MAC) value. The IV contributes *solely* to cryptographic security and the sequence number ensures *only* reliability of the message content. This is an important distinction as we delve further into the challenges of uTLS.

TLS supports a variety of ciphers, but we focus our work on a commonly-used cipher mode called Cipher Block Chaining (CBC) [8]. CBC mode has the property that the IV for a given record equals the *encrypted* data of the previous record. For example, after encrypting the first record but before sending it, the ciphersuite stores the data for use as the IV for the following record. This process repeats for each record forming a dependency "chain" throughout the entire communication stream.

As mentioned, the sequence number does not provide cryptographic security, but rather, ensures solely the integrity of the data after decryption. As a result, the sequence number is often a simple record counter, monotonically increasing and starting at one.

4.2 Challenges to uTLS

Given the nature of TLS described above, there are several challenges to enabling out-of-order delivery. We first present the challenges, and explain our solutions in the next section. The first challenge is identifying record boundaries. Each record begins with the special 5-byte TLS header, but the receiving host cannot simply scan for these headers and attempt decryption. Re-segmentation and the potential for encrypted content to match a header can cause "false headers".

The second challenge deals with decryption. Recall that in CBC mode, decryption requires the *previous* record for the IV, and if decrypting out-of-order, there is a gap in the byte stream for which data has not yet arrived. This means that the previous record's encrypted content is not available. In CBC mode there appears no way to decrypt records out-of-order due to the dependency chain between records.

The third challenge is similar to the second, but deals with the sequence number. The MAC calculation will not succeed without the correct sequence number. When delivering records out-of-order, there is a gap of contiguous data. It is difficult to know exactly *how many* records fill that gap. Without knowing the logical record number of a given record, the MAC calculation may fail.

The last challenge deals with failed decryption and authentication. In normal TLS, a failure terminates the connection, but this is because a failure in-order implies corruption somewhere in the system. Similarly, uTLS should terminate the connection for an authentication failure *in-order*. It should not terminate the connection for an out-of-order failure to allow for possible incorrect sequence numbers. This does not violate security because corrupted data will eventually be tried in-order and will fail. Figure 3 illustrates these challenges.

5. uTLS: ADDRESSING CHALLENGES

We now present our solutions to the challenges posed by out-of-order delivery in TLS connections.

5.1 Identifying Record Boundaries

Each TLS record has a 5-byte header that marks the start of a potential record. Before attempting decryption on the record, however, the receiver compares the length field of the header with the length of contiguous bytes succeeding the header in the received stream. If the specified length exceeds the available bytes, the record is skipped, and the receiver continues looking for potential records. In this way, the receiver avoids attempting decryption until the entire record has arrived. Furthermore, by checking the header-specified length against the TLS maximum and minimum record lengths, the receiver can rule out some false headers and prevent a failed decryption altogether. Not all failed decryptions can be avoided, however. It is possible for encrypted data to match a TLS header. These cases require a soft failure (Challenge 4).

5.2 Decryption without IV

The dependency chain of CBC mode prevents out-of-order decryption in the presence of a lost record because decrypting a given record requires the previous one. To surmount this problem, we prepend a record's IV to that record *prior* to transmission. The receiver then sets the IV of the cipher to the bytes preceeding the record header before decrypting. This change to the protocol eliminates the need for the previous record to arrive before the current record. Rather, the current record needs only the bytes representing its IV to successfully decrypt.

We note that this modification is a change to the wire format. Furthermore, it increases the bandwidth overhead. However, this change is only required for backwards compatibility to TLS version 1.0 and below [5]. TLS version 1.1 and above [6], explicitly sends the IV on the wire. Therefore, unordered decryption without the IV is only a challenge in TLS version 1.0 and below. Lastly, because TLS version 1.1 transmits each record's IV explicitly, uTLS incurs *zero* bandwidth overhead going forward.

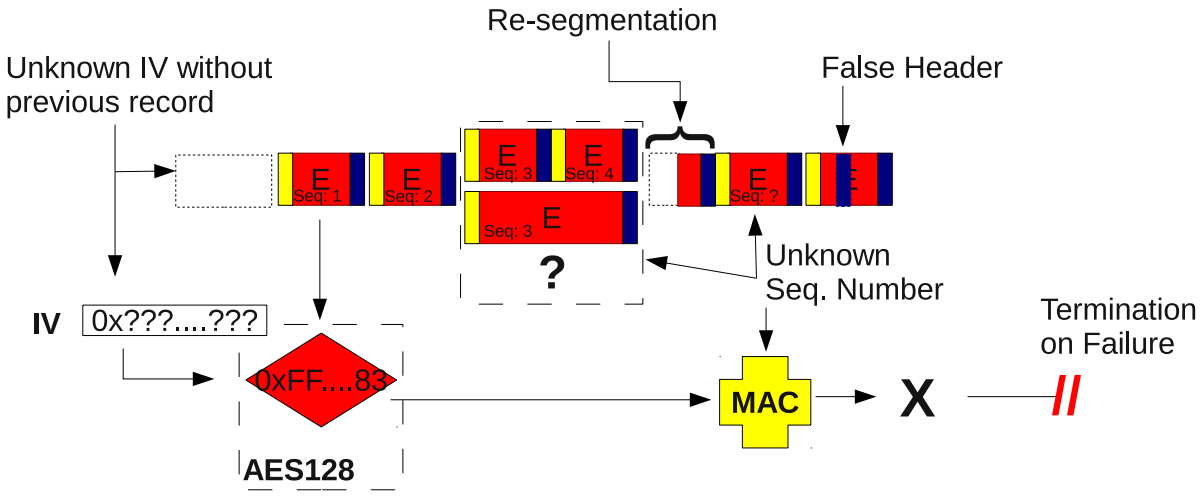

Figure 3: Challenges to uTLS.

It is important to note that TLS's security stems from the collective randomness and unpredictability of the cryptographic key and IV *pairing*. Explicitly sending the IV before a record header incurs no loss of security, though proving this is not the focus of this work.

5.3 MAC without Sequence Number

The TLS MAC provides message fidelity, but does not provide cryptographic security. As such, the sequence number that is used to authenticate decrypted records is not a secret; it increments by one for each record. We leverage this knowledge to *predict* sequence numbers for out-of-order records. Given a gap in the receive-queue of data that has not arrived yet, we can guess at the number of records contained in the gap. We then add this number to the last known successful sequence number used (ie. the record containing the cumulative ACK point). With a static record size, we predict with high accuracy the sequence number for a given out-of-order record.

Even with varying record sizes, we can try multiple sequence numbers in succession, provided a successful decryption using the key and IV. If the sequence number prediction fails to successfully authenticate a record out-of-order, the record is stored and will eventually be tried in-order. If this in-order MAC check fails, the connection is terminated due to corruption.

5.4 Soft Failure for Unordered Data

Because TLS only delivers data in-order, a decryption or authentication failure implies corruption. In uTLS, our out-of-order authentication may fail due to an incorrect sequence number prediction. uTLS's decryption almost always succeeds, due to the checking of the header-specified length. False headers *can* cause a decryption failure, though we did not observe this in our experiments.

Because of their differences, uTLS needs the ability to fail "softly" when out-of-order records fail. Addressing this issue is straightforward; we made a simple change to the TLS code on failed out-of-order records: scan the receive queue for other potential records, and, if no records are available, return zero to the application.

6. PERFORMANCE AND EVALUATION

We now present performance results of our uTLS implementation. While we believe further optimization of our uTLS code is possible, these results demonstrate the plausability of our approach with regards to CPU utilization and network bandwidth overhead.

6.1 Implementation Complexity

Our preliminary prototype of uTLS was adapted from the OpenSSL [15] open source implemention of the TLS protocol version 1.0. The uTCP kernel modification added 295 lines (2.3%) of kernel code, while our uTLS implementation contributed an additional 557 lines (1.8%) of user code to OpenSSL's `libssl` library, but does not modify the `libcrypto` library.

Our prototype focuses on minimizing the CPU utilization of the uTLS code, by reading from the out-of-order socket only when the application requests new data; rather than reading continuously from the socket and storing data in user space. This comes at a fundamental cost, however, to application performance in some scenarios because it may unnecessarily delay data that could be delivered in-order but arrives after some logically "later" data in the stream. We discuss this tradeoff in detail in Section 7.1.

Lastly, the performance of our prototype increased significantly by storing the locations of potential headers in the receive queue. This prevents unnecessarily re-checking for potential headers a second time through the queue.

6.2 Bandwidth and CPU Costs

As uTLS is a part of the larger Tng framework, we present the performance results for all of Tng's application-level record encoding and out-of-order delivery protocols, as implemented by our uTCP, uCOBS and uTLS prototype libraries. We emphasize that these user-space libraries represent only two of the many ways applications might utilize uTCP, and that these libraries were written with little emphasis on tuning or optimization.

Figure 4 compares the CPU processing cost and application-perceived throughput for COBS and TLS encoding/decoding, atop both standard TCP streams and uTCP streams (uCOBS, uTLS), at several loss rates. The experiment is a 30MB bulk transfer over a path with 60ms RTT. Figure 4(a) shows CPU time consumption, the lighter part of each bar representing user time and the darker part representing kernel time. Figure 4(b) shows transfer bandwidth achieved. All results are normalized to the results of the same experiment over "raw" TCP, with no application-level record encoding.

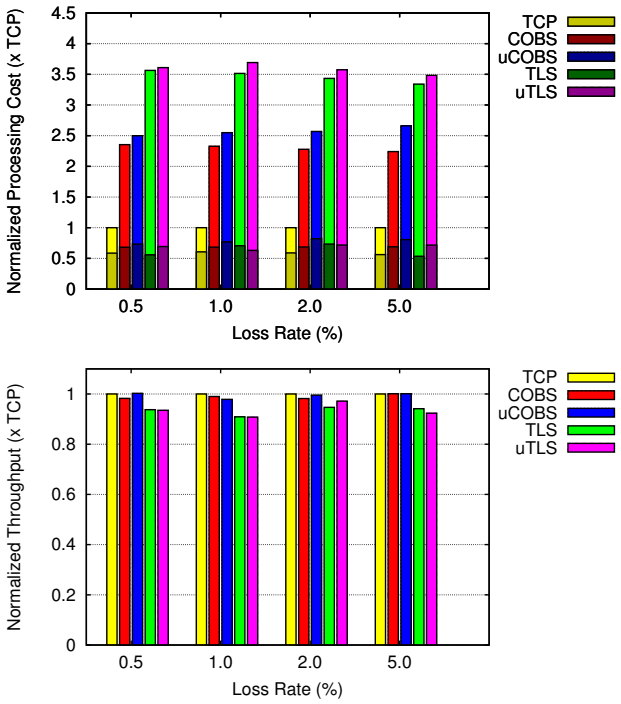

Figure 4: CPU and Throughput costs of using an application with TCP, COBS, uCOBS, TLS, and uTLS.

Unsurprisingly, kernel-level CPU consumption is barely affected by most of the application-level encodings. COBS encoding incurs some application-level processing cost, and TLS incurs more due to its encryption and authentication. Our uTLS prototype shows slightly higher utilization than TLS due to the necessity to scan for records and filter out false positives via the cryptographic MAC.

As shown in the throughput graph in Figure 4, the bandwidth penalty of uCOBS encoding (0.4% worst-case plus two marker bytes per application record) is essentially imperceptible. uTLS encoding incurs slightly more bandwidth overhead due to its addition of IVs to each record versus TLS version 1.0. However, as of TLS version 1.1, this cost is entirely attributable to the standard TLS encoding and is not affected by uTLS.

7. DISCUSSION AND FUTURE WORK

We discuss limitations of our current uTLS prototype, and how these issues might be addressed. Further, we discuss improvements and future work.

7.1 uTLS Performance Tradeoff

With the minimal overhead introduced by uTLS, we are confident that many real-time applications will benefit from unordered delivery. One such application is Voice-Over IP (VoIP). In this section, we discuss a homegrown application meant to demonstrate the effects of out-of-order delivery for a real-time application.

This application simulates a VoIP communication session by transmitting encoded audio frames at specified intervals, and playing them back at the receiving application based on the time received and the "jitter" buffer time window. For real-time applications, such as VoIP, the application's "perceived" performance, or quality, depends greatly on the latency. Because they are in-order protocols, TCP and TLS are theoretically ill-suited for such applications because a single dropped packet delays the delivery of all subsequent packets until successful retransmission. This "bursty loss" effect is in sharp contrast to an out-of-order protocol which does not delay any packets other than those that are dropped. Thus, with uTLS we expect to see shorter burst lengths of delayed packets, on average, compared to TLS.

We use the Speex Audio Codec [25] to encode and decode voice audio frames. We experimented with many different conditions, but Figure 5 demonstrates the current performance tradeoff uTLS faces. This experiment uses a 60ms RTT, 2.0% artificial network loss, and two jitter buffer sizes of 50ms and 120ms. Figure 5 shows a CDF of codec-perceived burst lengths of audio frames that miss their playback point at the application.

As expected, for the smaller jitter buffer of 50ms, roughly 0.87% x RTT, the average burst length for uTLS is less than that of TLS. However, as the jitter buffer size increases, we see TLS outperform uTLS. This is due to a limitation in the way the current uTLS prototype reads from the kernel's socket.

The uTLS prototype reads raw, encrypted data from the socket and stores it in-order in a linked list. In order to minimize CPU overhead for traversing and managing this list, data is only read from the socket when there are no complete records in the list. Once a packet is dropped, other later packets may be read from the socket into the linked list. Subsequent reads by the application will be serviced by the records in the linked list, even if the retransmitted packet

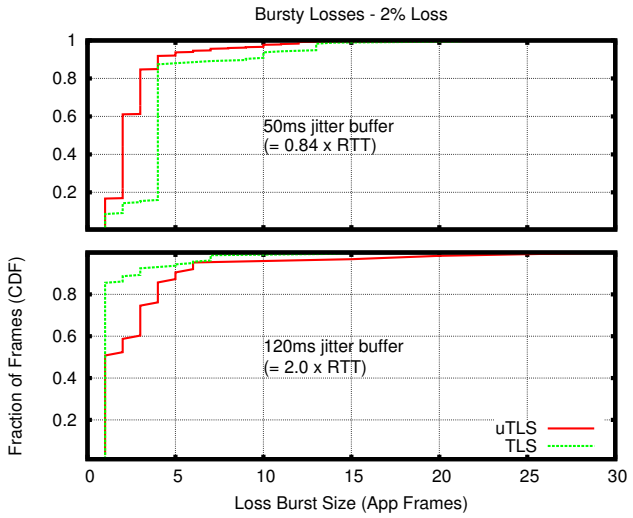

Figure 5: A comparison of uTLS and TLS for two jitter buffer sizes.

exists in the kernel's socket. This behavior has a tendency to "lock out" packets that are initially dropped by the network. This performance tradeoff limits only the effectiveness of our current prototype and is not an inherent fault of uTLS. We discuss future work to address this, and other challenges in the next subsection.

7.2 Future Work

Future work should investigate the use of uTCP and uTLS in real-time applications. In particular, the example VoIP application described above demonstrates that significant improvements are possible in the protoype implementation. Future work should find the optimal tradeoff between minimizing CPU overhead cost and favoring the delivery of in-order data.

Some potential applications for a uTLS and TLS comparison include VoIP and Video conferencing and chat applications, multi-streaming web applications, such as searching and shopping. Multi-streaming applications stand to gain significantly because out-of-order data on a single, multiplexed link may actually be in-order data for one of the constituent "logical" streams. This may enable, for example, parts of a webpage to display more quickly, improving the perceived latency of the connection.

8. CONCLUSION

All of the Internet transports designed since TCP, despite their diverse characteristics, embody a common recognition that many important applications can benefit from out-of-order delivery. None of these out-of-order transports except UDP, however, has surmounted the high barriers to entry that today's Internet effectively places on new protocols layered atop IP. Even applications such as VoIP that traditionally run on UDP are shifting to TCP tunneling for network compatibility reasons. Instead of ignoring or fighting this trend, we have demonstrated a small suite of protocols that can offer applications out-of-order delivery while maintaining strict wire-compatibility with TCP and even TLS.

In this paper, we presented uTLS and showed its feasability in terms of bandwidth overhead and CPU utilization. As a bonus, the source code changes required to the common TLS implementation are less than 3.7%. Future work will address the tradeoff between low utilization and per-packet latency. With uTLS and uTCP, we demonstrate the ability to offer datagram semantics, including out-of-order delivery, with the reachability guarantee of TLS and TCP.

9. REFERENCES

[1] SPDY: An Experimental Protocol For a Faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.
[2] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *INFOCOM*, Apr. 2006.
[3] S. Cheshire and M. Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, Sept. 1997.
[4] J. Davies. DirectAccess and the thin edge network. *Microsoft TechNet Magazine*, May 2009.
[5] T. Dierks and C. Allen. The TLS protocol version 1.0, Jan. 1999. RFC 2246.
[6] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.1, Apr. 2006. RFC 4346.
[7] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.
[8] M. Dworkin. Recommendation for block cipher modes of operation, Dec. 2001. NIST Special Publication 800-38A.
[9] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.
[10] B. Ford. Structured streams: a new transport abstraction. In *SIGCOMM*, Aug. 2007.
[11] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets-VII*, Oct. 2008.
[12] 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.
[13] S. Kent and K. Seo. Security architecture for the Internet protocol, Dec. 2005. RFC 4301.
[14] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), Mar. 2006. RFC 4340.
[15] The OpenSSL project. `http://www.openssl.org/`.
[16] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *HotNets-IX*, Oct. 2010.
[17] J. Postel. User datagram protocol, Aug. 1980. RFC 768.
[18] E. Rescorla. HTTP over TLS, May 2000. RFC 2818.
[19] E. Rescorla and N. Modadugu. Datagram transport layer security, Apr. 2006. RFC 4347.
[20] J. Rosenberg. UDP and TCP as the new waist of the Internet hourglass, Feb. 2008. Internet-Draft (Work in Progress).
[21] R. Stewart, ed. Stream control transmission protocol, Sept. 2007. RFC 4960.
[22] Transmission control protocol, Sept. 1981. RFC 793.
[23] W. W. Terpstra, C. Leng, M. Lehn, and A. P. Buchmann. Channel-based unidirectional stream protocol (CUSP). In *INFOCOM Mini Conference*, Mar. 2010.
[24] O. Titz. Why TCP over TCP is a bad idea, Apr. 2001. `http://sites.inka.de/bigred/devel/tcp-tcp.html`.
[25] J.-M. Valin. The speex codec manual version 1.2 beta 3, Dec. 2007. `http://www.speex.org/`.
[26] D. Velten, R. Hinden, and J. Sax. Reliable data protocol, July 1984. RFC 908.
[27] W3C. The websocket api (draft), 2011. `http://dev.w3.org/html5/websockets/`.