TurboTLS: TLS connection establishment with 1 less round trip

Carlos Aguilar-Melchor[1] Thomas Bailleux[1] Jason Goertzen[2] David Joseph[1] Douglas Stebila[2]

1: SandboxAQ, Palo Alto, USA
2: University of Waterloo

February 10, 2023

Abstract

We show how to establish TLS connections using one less round trip. In our approach, which we call TurboTLS, the initial client-to-server and server-to-client flows of the TLS handshake are sent over UDP rather than TCP. At the same time, in the same flights, the three-way TCP handshake is carried out. Once the TCP connection is established, the client and server can complete the final flight of the TLS handshake over the TCP connection and continue using it for application data. No changes are made to the contents of the TLS handshake protocol, only its delivery mechanism. We avoid problems with UDP fragmentation by using *request-based fragmentation*, in which the client sends in advance enough UDP requests to provide sufficient room for the server to fit its response with one response packet per request packet. Clients can detect which servers support this without an additional round trip, if the server advertises its support in a DNS HTTPS resource record. Experiments using our software implementation show substantial latency improvements. On reliable connections, we effectively eliminate a round trip without any noticeable cost. To ensure adequate performance on unreliable connections, we use lightweight packet ordering and buffering; we can have a client wait a very small time to receive a potentially lost packet (e.g., a fraction of the RTT observed for the first fragment) before falling back to TCP without any further delay, since the TCP connection was already in the process of being established. This approach offers substantial performance improvements with low complexity, even in heterogeneous network environments with poorly configured middleboxes.

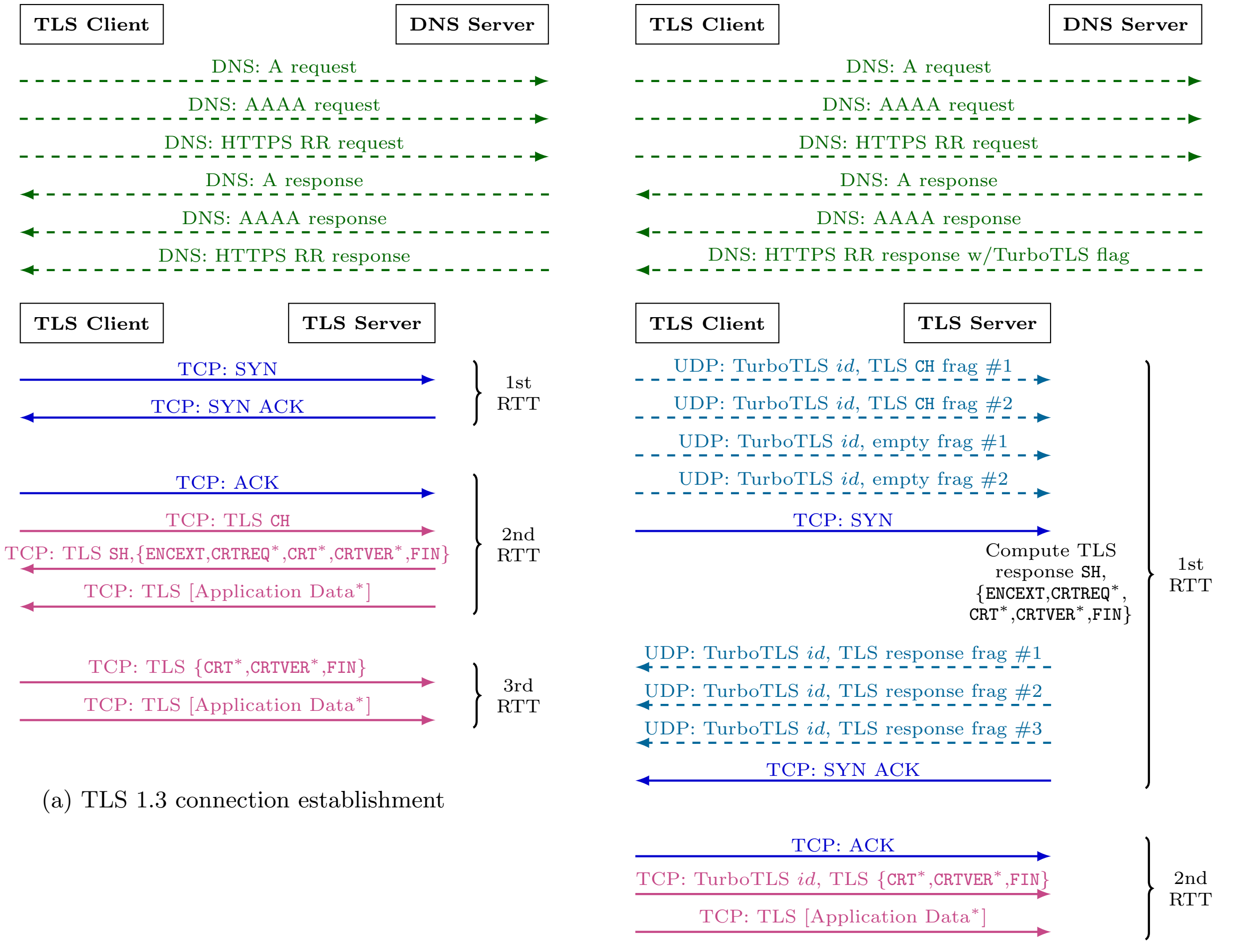
1 Introduction

The Transport Layer Security (TLS) protocol is ubiquitous and provides security services to many network applications. TLS runs over TCP. As shown in Figure 1a, the main flow for TLS 1.3 connection establishment [13] in a web browser is as follows.

First, the client makes a DNS query to translate the provided domain name into an IP address. Modern browsers simultaneously request from the DNS server an HTTPS resource record [20] which can provide additional information about the server's HTTPS configuration. Next, the client performs the TCP three-way handshake with the server. Once the TCP handshake has completed and a TCP connection is established, the TLS handshake can begin; it requires one client-to-server (C→S) flow and one server-to-client (S→C) flow before the client can start sending application data.

In total, excluding the DNS resolution, this results in two round trips before the client can send its first byte of application data (the TCP handshake and the first C→S and S→C flows of the TLS handshake), and another round trip before the client receives its first byte of response.

TLS does have a pre-shared key mode that allows for an abbreviated handshake permitting application data to be sent in the first C→S TLS flow, but this requires that the client and server have a pre-shared key in advance, established either through some out-of-band mechanism or saved from a previous TLS connection for session resumption.


TLS Client
DNS Server
DNS: A request
DNS: AAAA request
DNS: HTTPS RR request
DNS: A response
DNS: AAAA response
DNS: HTTPS RR response
TLS Client
TLS Server
TCP: SYN
TCP: SYN ACK
1st RTT
TCP: ACK
TCP: TLS CH
TCP: TLS SH,{ENCEXT,CRTREQ*,CRT*,CRTVER*,FIN}
TCP: TLS [Application Data*]
2nd RTT
TCP: TLS {CRT*,CRTVER*,FIN}
TCP: TLS [Application Data*]
3rd RTT
TLS Client
DNS Server
DNS: A request
DNS: AAAA request
DNS: HTTPS RR request
DNS: A response
DNS: AAAA response
DNS: HTTPS RR response w/TurboTLS flag
TLS Client
TLS Server
UDP: TurboTLS id, TLS CH frag #1
UDP: TurboTLS id, TLS CH frag #2
UDP: TurboTLS id, empty frag #1
UDP: TurboTLS id, empty frag #2
TCP: SYN
Compute TLS response SH, {ENCEXT,CRTREQ*, CRT*,CRTVER*,FIN}
UDP: TurboTLS id, TLS response frag #1
UDP: TurboTLS id, TLS response frag #2
UDP: TurboTLS id, TLS response frag #3
TCP: SYN ACK
1st RTT
TCP: ACK
TCP: TurboTLS id, TLS {CRT*,CRTVER*,FIN}
TCP: TLS [Application Data*]
2nd RTT

(a) TLS 1.3 connection establishment

(b) TurboTLS connection establishment

Figure 1: Comparing message flow of TLS 1.3 and TurboTLS. Legend: dashed line - UDP; solid line - TCP. * denotes optional message. $\{\ldots\}$ denotes messages encrypted using TLS handshake traffic secret and $[\ldots]$ denotes messages encrypted using TLS application traffic secret.

Our contributions. We describe a method, which we call TurboTLS, for removing one round trip of latency from TLS connection establishment by transmitting the first two flows of the TLS handshake over UDP while doing the TCP three-way handshake in parallel, then switching over to the TCP connection for the final C→S handshake flow and the transmission of application data. The message flow of TurboTLS is shown in Figure 1b. It allows the client to start sending its first byte of application in just one round trip (excluding the DNS resolution), without requiring any pre-shared key. TurboTLS does not require any change to the contents or state machine of the TLS protocol: it only changes the network delivery mechanism. We employ several techniques to make TurboTLS operate smoothly in a heterogeneous network environment where there may be UDP packet loss, where some servers may not support TurboTLS, and where intermediary network devices may have trouble with UDP fragmentation.

Table 1 summarizes the characteristics of TurboTLS compared with other relevant network security protocols; see Section 2 and Section 5.1 for more details.

2 Background

An application wishing to establish a secure connection between a client and a server will select a protocol, or combination of protocols across network layers, depending on a number of factors. Depending on the

	Runs over	**UDP 1 req. ⇒ 1 resp.**	**Provides conn.**	**Kernel netw.**	**No state**	**TLS-based**	**Widely deployed**	**RTT to 1st byte**
TLS 1.2	TCP	—	●	●	●	●	●	3
TLS 1.2 FalseStart	TCP	—	●	●	●	●	●	2
TLS 1.3	TCP	—	●	●	●	●	●	2
TLS 1.3 PSK	TCP	—	●	●	○	●	●	1
TLS 1.3 ECH	TCP	—	●	●	○	●	◓	1
OPTLS	TCP	—	●	●	○	●	○	1
TLS 1.3 + TCP Fast Open	TCP	—	●	●	○	●	◔	1
DTLS 1.3	UDP	●	○	●	●	●	●	2
QUIC	UDP	○	●	○	●	◕	◓	1
MinimaLT	UDP	●	●	○	●	○	○	1
MinimaLT with state	UDP	●	●	○	○	○	○	0
TurboTLS	UDP+TCP	●	●	◕	●	●	—	1
TurboTLS + PSK	UDP+TCP	●	●	◕	○	●	—	0
TurboTLS + ECH	UDP+TCP	●	●	◕	○	●	—	0

Table 1: Characteristics of TurboTLS compared to TLS and other optimized/accelerated protocols and variants.
Legend: ●: yes; ◕, ◓, ◔: partial; ○: no; —: not applicable. **Columns:** UDP 1 req. ⇒ 1 resp.: does each UDP request packet lead to at most one response packet? Provides conn.: does the protocol provide connection-oriented (reliable, in-order) transport to the application? Kernel netw.: are connection-oriented features generally implemented in the kernel? No state: can full optimization be achieved without pre-shared state between client and server? RTT to 1st byte: how many round trips required until the client can send its first application byte, including TCP 3-way handshake if necessary.

application (performance requirements, reliability requirements, etc.) there may be a preference for a connection-oriented or connectionless protocol. Finally, if a server does not support a protocol, or an initial request is blocked due to other reasons such as firewall filtering, then the client may need to fall back to another protocol that reaches the server and is supported.

In this section, we review several existing options to set up application-level secure channels, focusing on the TLS protocol, variants of TLS, and protocols aiming to replace TLS. Our first-level categorization is whether the protocol runs over TCP or UDP.

2.1 Secure channel protocols over TCP

Applications requiring connection-oriented communication typically run over TCP, such as 'vanilla' TLS. TCP uses an initial round trip to set up the connection, using the TCP three-way handshake, then a further round trip is needed to complete the TLS 1.3 handshake (or two further round trips for TLS 1.2 [15]), during which the cryptographic parameters are negotiated, session keys are exchanged, and authentication happens.

While the number of rounds trips and the resulting inherent latency is not always a problem for clients/servers in close proximity to one another, this presents a significant inconvenience where parties are far apart or suffer high network latency. To ameliorate this issue, a series of optimizations to TLS have been proposed, using a range of approaches.

Data-based optimizations. Some optimizations reduce the amount of data transmitted without reducing the number of round trips. Perhaps the simplest approach is that of Compact TLS [14], which changes the format of TLS handshake messages by removing obsolete fields and defining profiles of common options. Another light-touch optimization is TLS Cached Information Extension [18], which allows clients and servers to indicate they already have certain sets of values, such as intermediate certificates, to avoid re-transmitting them, which can save a significant amount of bandwidth, especially in the context of post-quantum cryptography which typically results in larger intermediate certificates [22].

Optimizations using previous state. Some optimizations are possible if the client has some prior server-dependent state, either from a previous connection or from some public directory.

TLS has a pre-shared key (PSK) mode in which a client can make use of a pre-shared symmetric key to save one round trip, allowing a client to start sending application data in the first TLS 1.3 PSK flow (and thus on the second C→S flow including the TCP three-way handshake).

TCP Fast Open [3] allows a client to save a cryptographic cookie from a previous TCP connection and use it in a subsequent TCP connection to immediately start sending application data without having to do a TCP three-way handshake on the subsequent connection. TLS running over TCP Fast Open would obviously then save one round trip.

OPTLS [7] was an alternative design for the TLS 1.3 handshake, running over TCP, which supported a so-called 0-RTT mode allowing for a client to send application in its first C→S TLS flow (and thus on the second C→S flow including the TCP three-way handshake) provided that the client had previously cached or obtained out-of-band the server's semi-static public key.

Encrypted Client Hello [16] is a proposed TLS extension that enables the client to encrypt more of the `Client Hello` message as well as send early application data in the first TLS C→S flight (and thus on the second C→S flow including the TCP three-way handshake) provided the client (similarly to in OPTLS) has previously cached or obtained out-of-band a public key of the server (which could be distributed in a DNS record).

There were also several other modifications to TLS 1.2 that made use of previous state, including TLS Snap Start [9] and "fast-track" client-side caching [21].

Other message flow optimizations. TLS False Start [10] is a modification to TLS 1.2 that allows the client to start sending application data one flight earlier. Note that, since TLS 1.2 had an extra round trip resulting in the client sending application data only in its third C→S flow, TLS False Start only moves TLS 1.2 up to parity with TLS 1.3 in terms of number of round trips before application data can be sent, and so we could also point out that TLS 1.3 itself represents an optimization of the TLS protocol.

2.2 Secure channel protocols over UDP

Another branch of optimizations utilizes the connectionless properties of UDP to fast-track performance. However, since many applications need to connection-oriented channels for data transmissions, most optimizations running on top of UDP specify their own procedures for packet reordering, packet loss, and session management, although we first briefly discuss DTLS, which does not.

Protocols not providing connection-oriented features to applications. DTLS [17] runs exclusively over UDP, including for transmission of application data. Because it runs on UDP, it is left to the application to reorder packets and deal with loss. Cryptographically, DTLS is based on TLS. DTLS can be particularly useful when trying to avoid problems such as TCP meltdown, whereby applications may be trying to transport TCP traffic inside a secure tunnel which also runs on TCP, essentially stacking TCP upon TCP and thereby amplifying the occurrence of TCP timeout and other related problems. For this reason DTLS is often used for VPN applications.

Protocols providing connection-oriented features to applications. MinimaLT [12] runs exclusively over UDP and uses a completely different protocol design compared to TLS, but has not, to date, seen widespread usage.

QUIC [6] is another approach. Designed originally to improve performance of encrypted transport for Google's internal services, QUIC is an ambitious and completely separate connection-oriented protocol running on top of UDP. Like MinimaLT, QUIC fundamentally merges the transport and security layers, and provides many other protocol-specific optimizations, such as providing varying header lengths (a longer header format is used for packets establishing connections), ACK-based packet loss detection which overcomes the instability of UDP by providing a grace period to in-flight packets, packet re-ordering, and others. A further benefit of QUIC is that for re-established connections, it is possible to send encrypted application data in the first packet by re-using previously agreed cryptographic parameters and utilizing a pre-shared key setup, using a technique similar to those of OPTLS, and again at the cost of forward secrecy for the initial data sent.

3 TurboTLS design

As described in Figure 1b, TurboTLS sends part of the TLS handshake over UDP, rather than TCP. Switching from TCP to UDP for handshake establishment means we cannot rely on TCP's features, namely connection-oriented, reliable, in-order delivery. However, since the rest of the connection will still run over TCP and only part of the handshake runs over UDP, we can reproduce the required functionality in a lightweight way without adding latency and allowing for a simple implementation.

Fragmentation. One of the major problems to deal with is that of fragmentation. TLS handshake messages can be too large to fit in a single packet – especially with long certificate chains or if post-quantum algorithms are used.

Obviously the client can fragment its first C→S flow across multiple UDP packets. To allow a server to link fragments received across multiple UDP requests, we add a 12-byte connection identifier field, containing a client-selected random value *id* that is used across all TurboTLS fragments sent by the client. The connection identifier is also included in the first message on the established TLS connection to allow the server to link together data received on the UDP and TCP connections. To allow the server to reassemble fragments if they arrive out-of-order, each fragment includes the total length of the original message as well as the offset of the current fragment; this can allow the server to easily copy fragments into the right position within a buffer as they are received.

Similarly, the server can fragment its first S→C flow across multiple UDP packets. One additional problem here however is that the S→C flow is typically larger than the C→S flow (as it typically contains one or more certificates), so the server may have to send more UDP response packets than UDP request packets. As noted by [24] in the context of DNSSEC, many network devices do not behave well when receiving multiple UDP responses to a single UDP request, and may close the port after the first packet, dropping the request. Subsequent packets received at a closed port lead to ICMP failure alerts, which can be a nuisance.

We employ a recent method proposed by Goertzen and Stebila [4] for DNSSEC: request-based fragmentation. In the context of large resource records in DNSSEC, [4] had the first response be a truncated response that included information about the size of the response, and then the client sent multiple additional requests, in parallel, for the remaining fragments. This ensured that there was only one UDP response for each UDP request. We adapt that method for TurboTLS: the client, in its first C→S flow, fragments its own C→S data across multiple UDP packets, and additionally sends (in parallel) enough nearly-empty UDP requests for a predicted upper bound on the number of fragments the server will need to fit its response. This preserves the model of each UDP request receiving a single UDP response, reducing the impact of misbehaving network devices and also reducing the potential for DDoS amplification attacks.

Reliability. UDP does not have reliable delivery, so packets may be lost. Since the first TurboTLS round-trip includes the TCP handshake, we can immediately fall back to TCP if a UDP packet is lost in either direction. This will induce a latency cost of however long the client decides to wait for UDP packets to arrive before giving up and assuming they were lost.

In an implementation, the client delay could be a fixed number of milliseconds, or could be variable depending on observed network conditions; this need not be fixed by a standard. We believe that in many cases a client delay of just 2ms after the TCP reply is received in the first round trip will be enough to ensure UDP responses are received a large majority of the time. In other words, by tolerating a potential 2ms of extra latency on $X\%$ of connections, we can save an entire round-trip on a large proportion $(100 - X\%)$ of the connections. This mechanic was not implemented in the experimental results presented here and constitutes future work.

Advertising support. To protect servers who do not support TurboTLS from being bombarded with unwanted UDP traffic, it would be preferable if clients only used TurboTLS with servers that they already know support it. Clients could cache this information from previous non-TurboTLS connections, but in fact we can do better. Even on the first visit to a server, we can communicate server support for TurboTLS to the client, without an extra round trip, using the HTTPS resource record in DNS [20]. Today when web browsers perform the DNS lookup for the domain name in question, they typically send three requests in parallel: an A query for an IPv4 address, an AAAA query for an IPv6 address, and a query for an HTTPS resource record [20]. Servers can advertise support for TurboTLS with an additional flag in the HTTPS resource record and clients can check for it without incurring any extra latency.

4 Experimental analysis

We implemented TurboTLS to compare its performance with vanilla TLS 1.3. Our preliminary proof-of-concept implements most of TurboTLS as described in Section 3, but not yet completely; see the "Limitations" paragraph below.

Libraries and cryptographic algorithms. Our implementation of TurboTLS is based on OpenSSL [26], using Open Quantum Safe fork of OpenSSL to provide support for post-quantum algorithms [25]. We take advantage of OpenSSL's BIO interface to have a fine control over the I/O operations, allowing us to transmit some messages over UDP instead of TCP.

In our experiments, we considered two cryptographic suites, where we varied the public key algorithms used:

- Elliptic curves: ECDSA signatures and ECDH ephemeral key exchange using the nistp256/secp256r1 curve.
- Post-quantum: Dilithium2 signatures [11], and Kyber-512 key exchange [19]. This suite results in both the C→S and S→C TLS handshake flows being fragmented.

In both cases, we used the same symmetric algorithms (AES-128 in Galois counter mode, SHA-256). We use a single self-signed certificate, in other words, a certificate chain of length 1.

Network. We used four network configurations:

- Local: The client and server are in the same data center of a cloud provider, with a ping time of 261 microseconds.
- Continental: The client was in a data centre in Paris, and the server was in a data centre in Belgium, within the same cloud provider network, joined by a network connection with an observed ping time of 4.9 milliseconds.
- Transcontinental1: The client was in a data centre in Paris, and the server was in a data centre in Oregon, within the same cloud provider network, joined by a network connection with an observed ping time of 133 milliseconds.
- Transcontinental2: The client was in a data centre in Paris, and the server was in a data centre in Australia, within the same cloud provider network, joined by a network connection with an observed ping time of 269 milliseconds.

The only source of latency we introduce is distance, but there are many other reasons for latency to be in the hundreds of milliseconds: the number of network hops (when changing between providers, or going to end users), the server load (which can provoke waiting queues over each round-trip), and the technology of the intermediate networks (IoT, 3G, etc.). In practice, global studies show[1] that, ignoring server load or end-user delays, median connections lead to RTTs mostly between 50 ms (west coast to west coast) and 200 ms (west coast to Europe), and 90th percentile connections lead to RTTs going over 500 ms or even 1000 ms [5].

We addressed servers with IP addresses so we did not incur any time for DNS resolution, and the client assumed the server supported TurboTLS without making any DNS HTTPS resource record query.

Machines. In all cases, the machines used were Linux x86_64 cloud servers with 4 cores (8 vcores taking into account HyperThreading) of an Intel Xeon E5-2696V4 Processor with 16 GB of RAM.

[1] CAIDA's Macroscopic Internet Topology Monitor `https://www.caida.org/catalog/software/walrus/rtt/`

Results. Figure 2 shows the results of the experiment across the two cryptographic suites and four network configurations, comparing the latencies of TLS 1.3 versus TurboTLS. The results reported show latencies at the 50th percentile (median), 90th percentile, and 99th percentile, for a 10 second experiment. The results are presented as a table in the appendix with precise figures usable when a fine-grained comparison is needed.

As expected, in long distance connections (see Figure 2c, Figure 2d), where latency is primarily due to the time for information to travel between endpoints, saving one round-trip basically halves the latency.

In low-latency connections (see Figure 2a, Figure 2b) the situation is more complex. For traditional elliptic curve cryptography, the latency is reduced by either the observed RTT or in some cases by a slightly larger value. This is amplified in the post-quantum setting where latency reduction goes well beyond one RTT. Indeed, in the Paris–Belgium setting we observe a reduction in median values from 11ms to 5ms (RTT being 5ms), and in 99th percentile values the reduction is from 13ms to 5ms. In the Paris–Paris setting, all latencies (1.5 ms median, 1.9 ms 90th, and 2.5 ms 99th percentile) are halved, with a decrease that is up to six times the observed RTT (0.2 ms). Our experiments seem to indicate that the observed phenomenon is linked to the increase in the number of fragments that need to be handled (the reduction of latency increases the handshakes per second, and using post-quantum algorithms also increases the number of fragments). In other words, having a UDP thread that polls, gathers, reconstructs and hands-over handshakes to a set of TCP threads, seems to be more efficient that having each TCP thread concurrently poll, gather and reconstruct their own stream. Further experiments are planned to confirm this hypothesis.

The end result is that, in practice, for classical cryptography, one observes the expected reduction in latency, one RTT, or slightly more in a local setting. For post quantum cryptography, latencies are roughly halved in all the experimental settings, over long distance connections where distance is the main contributor to latency, but also, unexpectedly, over short distance connections, where one RTT is much smaller than half the latency.

Limitations. Our implementation does not include, as of yet, the request-based fragmentation feature (in other words, the client only sends fragments for its actual packets, and the server responds with with as many UDP packets as needed). It also does not, as yet, fall back to TCP if the UDP packets are lost or delayed too much but, in our handful of trials, we never observed a failure of TurboTLS because of such issues (or any other reason). We plan to address these limitations with an updated implementation which will be reported in a revision of this paper.

5 Discussion

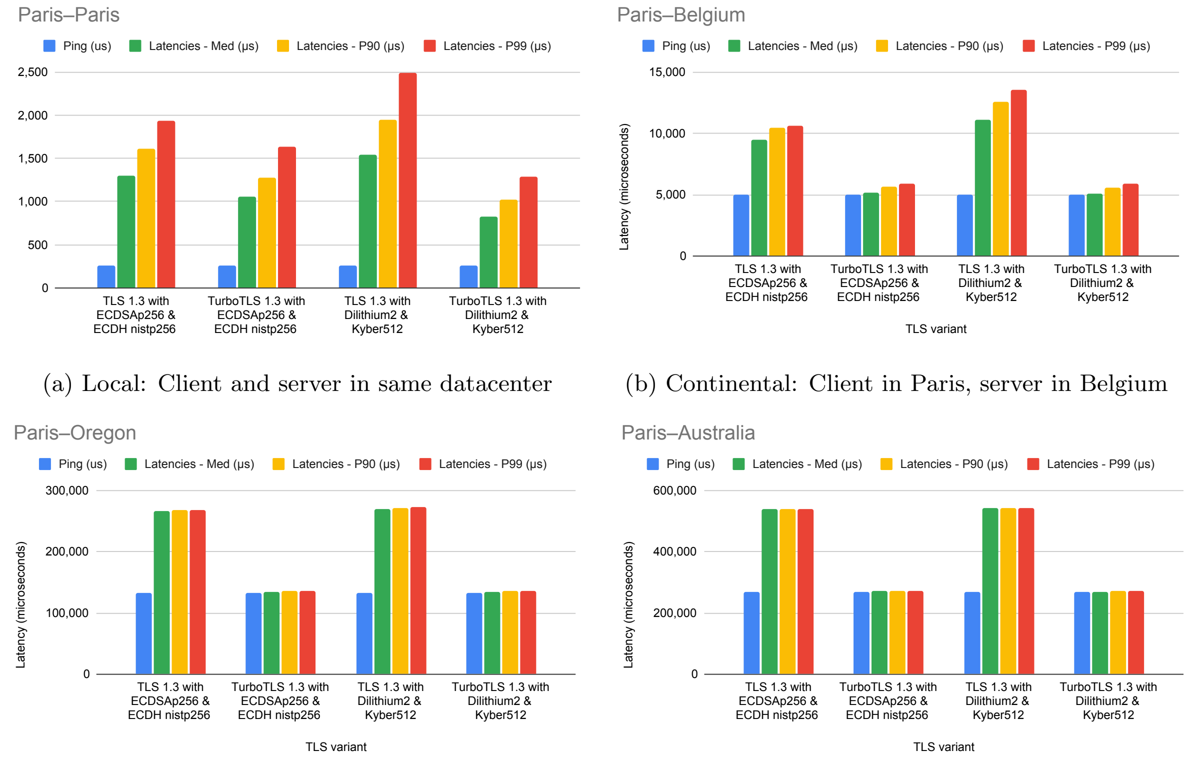
5.1 Comparison with other protocols

The protocols presented in Section 2 fall in one or more of the following categories:

- doing more than one round trip (TLS 1.2, TLS 1.2 FalseStart, TLS 1.3, DTLS 1.3, Compact TLS and TLS Cached Information Extension);
- modifying TCP/UDP directly or modify the way TCP/UDP are expected to be used (TLS 1.3 + TCP Fast Open, DTLS, MinimaLT, QUIC); or
- maintaining a state (TLS 1.3 + TCP Fast Open, TLS 1.3 PSK, OPTLS, TLS Encrypted Client Hello).

TurboTLS requires one round trip, uses TCP and UDP without modifying them, and as middleboxes would expect them to be used, and does not require a state. The rest of this section is dedicated to explain the drawbacks of falling into one of the three categories above.

One round trip. Doing more than one round trip increases latency by at least one RTT. As already noted, global studies provided to the community by CAIDA show that the RTT for median connections is between 50 and 200ms, mainly due to hops and distance, and if we consider the 90th percentile of connections, RTTs are beyond 500ms. This latency introduction is amplified by an integer factor for protocols in which connections occur iteratively (e.g. get a web page, get frames in the page, get images in the frames). Besides user experience, this also has an impact on usual implementations in which a server thread from a pool



(a) Local: Client and server in same datacenter

(b) Continental: Client in Paris, server in Belgium

(c) Transcontinental: Client in Paris, server in Oregon

(d) Transcontinental: Client in Paris, server in Australia

Figure 2: Comparing performance of TLS 1.3 and TurboTLS in 4 network settings with 2 cryptographic suites (elliptic curves or post-quantum). Latencies reported are time in microseconds from start of connection establishment until client sends its first byte of application data.

will not come back to the pool until it finishes dealing with a client. For connections with an RTT over a few milliseconds, when replacing TLS 1.3 by TurboTLS, we indeed observed a multiplication by two of the maximum handshakes per second that could be handled by a simple, yet usual, server implementation (thread pool with as many threads as cores that asynchronously handles connections). In future work we will extend our tests to high-end TLS reverse-proxies under heavy load, but the initial results are encouraging.

Standard and expected usage of TCP/UDP. Some protocols modify TCP/UDP or the way they are expected to be used. Of course, TCP Fast Open modifies TCP itself by introducing cookies for the first flight (which also requires a state). This probably explains why, even if the initial proposal is from 2011, it is still an experimental RFC and not enabled by default on most browsers. Other protocols, like DTLS, MinimaLT or QUIC, just use UDP, without modifying it, but do long-term bidirectional exchanges, which is not the usual for UDP. Long-term bidirectional exchanges are in general done over TCP and most protocols using UDP either follow the one query/one reply model (e.g. DNS) or the one query/many replies model (e.g. FTP download).

Using UDP for long-term bidirectionnal exchanges introduces two issues: instability and computational overhead. The main and simplest reason for UDP instability are firewall rules which often block such traffic, except for the one query/one reply model. Besides that, some middlebox functions for long term bidirectional traffic are only available for TCP and with UDP will either reduce performance or cause instability. RFC9312 [8] provides extensive guidance on how to deal with usual traffic management over QUIC: "Passive Network Performance Measurement and Troubleshooting; Stateful Treatment of QUIC Traffic; Address Rewriting to Ensure Routing Stability; Server Cooperation with Load Balancers; Filtering Behavior; UDP Blocking, Throttling, and NAT Binding; DDoS Detection and Mitigation; Quality of Service Handling and ECMP Routing;” and more. UDP bidirectional long-term exchanges that run through middleboxes that have not implemented these features (adapted to QUIC or to other protocols) will suffer from a lower quality of service and stability, sometimes with catastrophic effects [2]. On top of that, when connection-oriented features are provided (e.g. by QUIC or MinimaLT), one general drawback is that the implementation of the protocol needs to provide for packet reordering and recovery from packet loss, in user-space, whereas protocols running over TCP receive that for free from the operating system’s kernel-space TCP implementation, which has typically been highly tuned over many years, leading among other things to fewer interrupts and copies.

No state. Maintaining a state brings obvious issues (no benefit on first connection, lifetime, complexity) but most importantly, in the case of TLS, it also induces in general a loss of forward secrecy and thus of security. TLS 1.3 in pre-shared key (PSK) mode gains one round-trip only when it completely drops forward secrecy (as it relies on a pre-shared secret). This is somewhat mitigated by OPTLS and the Encrypted Client Hello TLS extension: if the client has previously obtained the server public key, then use of OPTLS and Encrypted Client Hello TLS extension are improved by one round-trip, and the loss of forward secrecy only affects the first flow of messages from the client to the server. The rest of the communication has forward secrecy.

No TLS changes. Additionally, TurboTLS shares, with TLS 1.3 + TCP Fast Open, another nice feature: TurboTLS makes no change whatsoever to the content of a TLS handshake, only changes the delivery mechanism. As a result, all cryptographic properties of TLS are untouched. In fact, it is possible to implement TurboTLS without changing the client or server’s TLS library at all, and instead use transparent proxies on both the client and server side to change the network delivery from pure TCP in TLS to UDP+TCP in TurboTLS. Of course in such a construction the initial client or server, who does not know TurboTLS, will observe two round trip times, but if each proxy is close to its host (say on the same machine), then the two round trip times will be negligible, and the higher latency client–server distance will only be covered over one round trip.

5.2 Denial-of-Service (DoS) considerations

We now consider the implications for TurboTLS of various types of denial-of-service and distributed denial-of-service attacks, including whether a TurboTLS server is a victim in a DoS attack or being leverage by attacker to induce a DDoS attack elsewhere. TurboTLS runs on top of both TCP and UDP so we have to consider attacks involving both protocols.

DoS attacks on TurboTLS servers. The most significant TCP DoS attack is the SYN flood attack where a target machine is overwhelmed by TCP SYN messages faster than it can process them. This is because a server, upon receiving a SYN, typically stores the source IP, TCP packet index number, and port in a ‘SYN queue’, and this represents a half-open connection. An attacker could flood the server with SYN messages thereby exhausting its memory. The server cannot just arbitrarily drop connections because then legitimate users may find themselves unable to connect. There are many protections against SYN flood attacks, one of which is allocating only very small amounts (micro blocks) of memory to half-open connections. Another is using TCP cryptographic cookies [1, 23] whereby the sequence number of the ACK encodes information about the SYN queue entry so that the server can reconstruct the entry even if it was not stored due to having a full SYN queue. TCP cookies enjoy support in the Linux kernel – this and other such mitigations are already sufficient to protect TurboTLS from SYN floods.

In general there are several vectors to consider for resource exhaustion attacks on a server running TurboTLS. The server needs to maintain a buffer of received UDP packets containing fragments of a TLS `Client Hello` message. To avoid memory exhaustion attacks, a server can safely bound the memory allocated to this buffer and flush old entries on a regular basis (e.g., after two seconds). In the worst case, a legitimate client whose UDP packets are rejected from a busy server or flushed early will be able to fall back to vanilla TLS over TCP, and will incur negligible latency loss (compared to TLS over TCP) in doing so, because TurboTLS starts the TCP handshake in parallel to the first C→S UDP flow. An attacker spoofing IP addresses and sending well-formed `Client Hello` messages could also try to exhaust a server’s CPU resources by causing a large amount of cryptographic computation. Again, a server under attack can limit the CPU resources allocated to UDP-received `Client Hello` messages, and then fall back to vanilla TLS over TCP. In the worst case, legitimate clients affected by this and having to fall back to vanilla TLS over TCP will incur negligible latency loss compared to TLS over TCP since the TCP handshake has already been started in parallel.

DDoS attacks leveraging TurboTLS servers. UDP reflection attacks present another threat. Typical defenses against these are blocking unused ports, rate limiting based on expected traffic loads from peers (exorbitant traffic loads are likely to be malicious), or blocking IPs of other known vulnerable servers. However such defenses are provided by middleboxes and therefore do not affect the protocol.

It should be noted here that the redundant UDP packets sent along with `Client Hello` are part of the TurboTLS-specific technique we call request-based-fragmentation to mitigate *against* a client's middlebox defenses incorrectly filtering TurboTLS connections, as otherwise multiple UDP responses to a single UDP request could be flagged as malicious behaviour. Furthermore, the one-to-oneness of the UDP request/response significantly reduces the impact of any amplification attack which tries to utilize a TurboTLS server as a reflector: an attacker would have to send one UDP packet for every reflected packet generated by the server, meaning that initial requests and responses are of comparable sizes, making the amplification factor so low that it would be an ineffective use of resources. Furthermore, the UDP requests ultimately must contain a fully formed `Client Hello` before the server responds, limiting the amplification factor.

5.3 TurboTLS improvements

We briefly mention a few alternative TurboTLS designs that may improve compatibility or can further reduce the latency assuming pre-shared state, and which may be interesting as future work.

TurboTLS optimization: TurboTLS for TLS in pre-shared key mode. Pre-shared key (PSK) mode of TLS 1.3 allows a client and server with a pre-shared symmetric key to eliminate parts of the handshake, and allows the client to optionally start sending encrypted application data its first C→S TLS flow, albeit without forward secrecy. The TurboTLS technique could be applied to TLS 1.3 PSK mode, running the first TLS 1.3 PSK C→S and S→C flows (including any early application data) over UDP and then switching over to TCP for the rest of the connection. This would allow for transmission of application data on the very first C→S TurboTLS flow, but comes at the cost of sacrificing forward secrecy, since PSK mode does not offer it. Early application data in both the first C→S and first S→C flows would be over UDP, with only the lightweight reliability features offered by TurboTLS compared to the more extensive reliability features offered by TCP.

TurboTLS optimization: TurboTLS + TLS encrypted client hello. Encrypted client hello (ECH) [16] is a mechanism to encrypt parts of the TLS 1.3 handshake under a semi-static server public key. This mechanism even allows for the transmission of application data one round trip earlier, but only by sacrificing forward secrecy. The TurboTLS approach combined with ECH could allow for transmission of application data on the very first C→S TurboTLS flow, at the cost of sacrificing forward secrecy. Again, early application data flows would be over UDP with TurboTLS's lightweight reliability features compared to TCP's more extensive reliability.

TurboTLS variant: UDP first stage + TLS 1.3 PSK handshake. When the UDP and TCP payloads of TurboTLS are combined, they contain an unaltered TLS 1.3 handshake. However, if the TCP portion is inspected on its own, it will appear to be only a part of a TLS handshake, and there is the potential that this could cause compatibility problems for some middleboxes/firewalls/interceptors. An alternative would be for the TLS handshake to terminate after the UDP portion of TurboTLS is completed, use the TLS keying material exporter paradigm to output a shared secret between the client and the server, and then use that shared secret as a pre-shared key in a TLS 1.3 PSK mode handshake over the TCP connection. This still maintains the RTT and latency improvements offered by TurboTLS, but ensures that the data within the TCP payloads are a fully standards-compliant TLS 1.3 PSK handshake transcript, which should further reduce the risk of incompatibilities from poorly configured middleboxes. (Note this differs from the

"TurboTLS optimization: TurboTLS for TLS in pre-shared key mode" mentioned above: the earlier paragraph on optimization for TLS in PSK mode is about using the TurboTLS technique to split a non-forward secure PSK handshakes across UDP and TCP, whereas this paragraph's TurboTLS variant does a forward-secure handshake in the UDP first stage and then uses the output of that as a PSK in a TLS 1.3 PSK handshake.)

Acknowledgements

D.S. and J.G. were supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grant RGPIN-2022-03187. The authors want to thank Adrien Guinet for his help with code reviewing and optimization.

A Experimental results

Protocol Mode	**Signature & Key Exchange**	**Ping**	**Throughput**		**Latency**	
				Median	**P90**	**P99**
		μs	hs/sec	μs	μs	μs
Paris–Paris						
TLS 1.3	ECDSAp256 & ECDH nistp256	261	2,901	1,299	1,608	1,935
TurboTLS 1.3	ECDSAp256 & ECDH nistp256	261	3,461	1,057	1,274	1,634
TLS 1.3	Dilithium2 & Kyber512	261	2,478	1,546	1,946	2,497
TurboTLS 1.3	Dilithium2 & Kyber512	261	4,400	821	1,024	1,290
Paris–Belgium						
TLS 1.3	ECDSAp256 & ECDH nistp256	4,979	421	9,507	10,493	10,647
TurboTLS 1.3	ECDSAp256 & ECDH nistp256	4,979	745	5,207	5,650	5,893
TLS 1.3	Dilithium2 & Kyber512	4,979	357	11,131	12,565	13,583
TurboTLS 1.3	Dilithium2 & Kyber512	4,979	776	5,046	5,538	5,885
Paris–Oregon						
TLS 1.3	ECDSAp256 & ECDH nistp256	133,021	16	266,984	268,386	268,540
TurboTLS 1.3	ECDSAp256 & ECDH nistp256	133,021	32	133,981	135,450	135,583
TLS 1.3	Dilithium2 & Kyber512	133,021	16	269,477	271,276	272,534
TurboTLS 1.3	Dilithium2 & Kyber512	133,021	32	133,764	135,781	135,835
Paris–Australia						
TLS 1.3	ECDSAp256 & ECDH nistp256	269,478	8	540,036	540,370	540,464
TurboTLS 1.3	ECDSAp256 & ECDH nistp256	269,478	16	270,276	270,506	270,792
TLS 1.3	Dilithium2 & Kyber512	269,478	8	542,831	543,375	543,488
TurboTLS 1.3	Dilithium2 & Kyber512	269,478	16	270,154	270,414	271,569

References

[1] D. J. Bernstein. SYN cookies. URL: `http://cr.yp.to/syncookies.html`.

[2] S. Chaudhary, P. Sachdeva, A. Mondal, S. Chakraborty, and M. Maity. YouTube over Google's QUIC vs internet middleboxes: A tug of war between protocol sustainability and application QoE, 2022. URL: `https://arxiv.org/abs/2203.11977`.

[3] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. RFC 7413, Dec. 2014. `doi:10.17487/RFC7413`.

[4] J. Goertzen and D. Stebila. Post-quantum signatures in DNSSEC via request-based fragmentation. arXiv, Nov. 2022. `doi:10.48550/ARXIV.2211.14196`.

[5] B. Huffaker, D. Plummer, D. Moore, and K. Claffy. Topology discovery by active probing. In *Symposium on Applications and the Internet (SAINT) 2002 Workshops*, pages 90–96. IEEE, 2002.

[6] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. doi:10.17487/RFC9000.

[7] H. Krawczyk and H. Wee. The OPTLS protocol and TLS 1.3. In *IEEE European Symposium on Security and Privacy (EuroS&P) 2016*, pages 81–96. IEEE, 2016. doi:10.1109/EuroSP.2016.18.

[8] M. Kühlewind and B. Trammell. Manageability of the QUIC Transport Protocol. RFC 9312, Sept. 2022. URL: https://www.rfc-editor.org/info/rfc9312, doi:10.17487/RFC9312.

[9] A. Langley. Transport Layer Security (TLS) Snap Start. Internet-Draft draft-agl-tls-snapstart-00, Internet Engineering Task Force, June 2010. URL: https://datatracker.ietf.org/doc/draft-agl-tls-snapstart/00/.

[10] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. RFC 7918, Aug. 2016. doi:10.17487/RFC7918.

[11] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[12] W. M. Petullo, X. Zhang, J. A. Solworth, D. J. Bernstein, and T. Lange. MinimaLT: minimal-latency networking through better security. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 2013*, pages 425–438. ACM Press, Nov. 2013. doi:10.1145/2508859.2516737.

[13] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018. doi:10.17487/RFC8446.

[14] E. Rescorla, R. Barnes, H. Tschofenig, and B. M. Schwartz. Compact TLS 1.3. Internet-Draft draft-ietf-tls-ctls-06, Internet Engineering Task Force, July 2022. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-ctls/06/.

[15] E. Rescorla and T. Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008. doi:10.17487/RFC5246.

[16] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-15, Internet Engineering Task Force, Oct. 2022. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-esni/15/.

[17] E. Rescorla, H. Tschofenig, and N. Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147, Apr. 2022. doi:10.17487/RFC9147.

[18] S. Santesson and H. Tschofenig. Transport Layer Security (TLS) Cached Information Extension. RFC 7924, July 2016. doi:10.17487/RFC7924.

[19] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, and D. Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[20] B. M. Schwartz, M. Bishop, and E. Nygren. Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs). Internet-Draft draft-ietf-dnsop-svcb-https-11, Internet Engineering Task Force, Oct. 2022. URL: https://datatracker.ietf.org/doc/draft-ietf-dnsop-svcb-https/11/.

[21] H. Shacham, D. Boneh, and E. Rescorla. Client-side caching for TLS. *ACM Trans. Inf. Syst. Secur.*, 7(4):553–575, Nov. 2004. doi:10.1145/1042031.1042034.

[22] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis. Post-quantum authentication in TLS 1.3: A performance study. In *NDSS 2020*. The Internet Society, Feb. 2020.

[23] W. A. Simpson. TCP Cookie Transactions (TCPCT). RFC 6013, Jan. 2011. doi:10.17487/RFC6013.

[24] L. Song and S. Wang. ATR: Additional Truncation Response for Large DNS Response. Internet-Draft draft-song-atr-large-resp-03, Internet Engineering Task Force, Mar. 2019. URL: https://datatracker.ietf.org/doc/draft-song-atr-large-resp/03/.

[25] D. Stebila and M. Mosca. Post-quantum key exchange for the Internet and the Open Quantum Safe project. In R. Avanzi and H. Heys, editors, *Selected Areas in Cryptography (SAC) 2016*, volume 10532 of *LNCS*, pages 1–24. Springer, Oct. 2017. URL: https://openquantumsafe.org/.

[26] The OpenSSL Project. OpenSSL version 1.1.1s, Nov. 2022. URL: https://www.openssl.org.