Workgroup: WEBTRANS
Internet-Draft:
draft-ietf-webtrans-overview-04
Published: 11 July 2022
Intended Status: Standards Track
Expires: 12 January 2023
Authors: V. Vasiliev
Google

The WebTransport Protocol Framework

Abstract

The WebTransport Protocol Framework enables clients constrained by the Web security model to communicate with a remote server using a secure multiplexed transport. It consists of a set of individual protocols that are safe to expose to untrusted applications, combined with a model that allows them to be used interchangeably.

This document defines the overall requirements on the protocols used in WebTransport, as well as the common features of the protocols, support for some of which may be optional.

Note to Readers

Discussion of this draft takes place on the WebTransport mailing list (webtransport@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/search/?email_list=webtransport>.

The repository tracking the issues for this draft can be found at <https://github.com/ietf-wg-webtrans/draft-ietf-webtrans-overview/issues>. The web API draft corresponding to this document can be found at <https://wicg.github.io/web-transport/>.

Status of This Memo

Table of Contents

1. Introduction

The WebTransport Protocol Framework enables clients constrained by the Web security model to communicate with a remote server using a secure multiplexed transport. It consists of a set of individual protocols that are safe to expose to untrusted applications, combined with a model that allows them to be used interchangeably.

This document defines the overall requirements on the protocols used in WebTransport, as well as the common features of the protocols, support for some of which may be optional.

1.1. Background

Historically, web applications that needed a bidirectional data stream between a client and a server could rely on WebSockets

[RFC6455], a message-based protocol compatible with the Web security model. However, since the abstraction it provides is a single ordered stream of messages, it suffers from head-of-line blocking (HOLB), meaning that all messages must be sent and received in order even if they are independent and some of them are no longer needed. This makes it a poor fit for latency-sensitive applications which rely on partial reliability and stream independence for performance.

One existing option available to Web developers are WebRTC data channels [RFC8831], which provide a WebSocket-like API for a peer-to-peer SCTP channel protected by DTLS. In theory, it is possible to use it for the use cases addressed by this specification. However, in practice, its use in non-browser-to-browser settings has been quite low due to its dependency on ICE (which fits poorly with the Web model) and userspace SCTP (which has very few implementations available).

An alternative design would be to open multiple WebSocket connections over HTTP/3 [I-D.ietf-httpbis-h3-websockets] in a manner similar to how they are currently layered over HTTP/2 [RFC8441]. That would avoid head-of-line blocking and provide an ability to cancel a stream by closing the corresponding WebSocket object. However, this approach has a number of drawbacks, which all stem primarily from the fact that semantically each WebSocket is a completely independent entity:

* Each new stream would require a WebSocket handshake to agree on application protocol used, meaning that it would take at least one RTT to establish each new stream before the client can write to it.

* Only clients can initiate streams. Server-initiated streams and other alternative modes of communication (such as the QUIC DATAGRAM frame [I-D.ietf-quic-datagram]) are not available.

* While the streams would normally be pooled by the user agent, this is not guaranteed, and the general process of mapping a WebSocket to a server is opaque to the client. This introduces unpredictable performance properties into the system, and prevents optimizations which rely on the streams being on the same connection (for instance, it might be possible for the client to request different retransmission priorities for different streams, but that would be much more complex unless they are all on the same connection).

The WebTransport protocol framework avoids all of those issues by letting applications create a single transport object that can contain multiple streams multiplexed together in a single context

(similar to SCTP, HTTP/2, QUIC and others), and can be also used to send unreliable datagrams (similar to UDP).

1.2. Conventions and Definitions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

WebTransport is a framework that aims to abstract away the underlying transport protocol while still exposing a few key transport-layer aspects to application developers. It is structured around the following concepts:

WebTransport session: A WebTransport session is a single communication context established between a client and a server. It may correspond to a specific transport-layer connection, or it may be a logical entity within an existing multiplexed transport-layer connection. Transport sessions are logically independent from one another even if some sessions can share an underlying transport-layer connection.

WebTransport protocol: A WebTransport protocol is a specific protocol that can be used to establish a WebTransport session.

Datagram: A datagram is a unit of transmission that is treated atomically.

Stream: A stream is a sequence of bytes that is reliably delivered to the receiving application in the same order as it was transmitted by the sender. Streams can be of arbitrary length, and therefore cannot always be buffered entirely in memory. WebTransport protocols and APIs are expected to provide partial stream data to the application before the stream has been entirely received.

Message: A message is a stream that is sufficiently small that it can be fully buffered before being passed to the application. WebTransport does not define messages as a primitive, since from the transport perspective they can be simulated by fully buffering a stream before passing it to the application. However, this distinction is important to highlight since some of the similar protocols and APIs (notably WebSocket [RFC6455]) use messages as a core abstraction.

Server: A WebTransport server is an application that accepts incoming WebTransport sessions.

Client:
A WebTransport client is an application that initiates the transport session and may be running in a constrained security context, for instance, a JavaScript application running inside a browser.

User agent: A WebTransport user agent is a software system that has an unrestricted access to the host network stack and can create transports on behalf of the client.

2. Common Transport Requirements

Since clients are not necessarily trusted and have to be constrained by the Web security model, WebTransport imposes certain requirements on any specific protocol used.

All WebTransport protocols MUST use TLS [RFC8446] or a semantically equivalent security protocol (for instance, DTLS [I-D.ietf-tls-dtls13]). The protocols SHOULD use TLS version 1.3 or later, unless they aim for backwards compatibility with legacy systems.

All WebTransport protocols MUST require the user agent to obtain and maintain explicit consent from the server to send data. For connection-oriented protocols (such as TCP or QUIC), the connection establishment and keep-alive mechanisms suffice. STUN Consent Freshness [RFC7675] is another example of a mechanism satisfying this requirement.

All WebTransport protocols MUST limit the rate at which the client sends data. This SHOULD be accomplished via a feedback-based congestion control mechanism (such as [RFC5681] or [RFC9002]).

All WebTransport protocols MUST support simultaneously establishing multiple sessions between the same client and server.

All WebTransport protocols MUST prevent clients from establishing transport sessions to network endpoints that are not WebTransport servers.

All WebTransport protocols MUST provide a way for the user agent to indicate the origin [RFC6454] of the client to the server.

All WebTransport protocols MUST provide a way for a server endpoint location to be described using a URI [RFC3986]. This enables integration with various Web platform features that represent resources as URIs, such as Content Security Policy [CSP].

3. Session Establishment

WebTransport session establishment is an asynchronous process. A session is considered *ready* from the client's perspective when the server has confirmed that it is willing to accept the session with the provided origin and URI. WebTransport protocols MAY allow clients to send data before the session is ready; however, they MUST NOT use mechanisms that are unsafe against replay attacks without an explicit indication from the client.

4. Transport Features

All transport protocols MUST provide datagrams, unidirectional and bidirectional streams in order to make the transport protocols interchangeable.

4.1. Session-Wide Features

Any WebTransport protocol SHALL provide the following operations on the session:

establish a session Create a new WebTransport session given a URI [RFC3986] and the origin [RFC6454] of the requester.

terminate a session Terminate the session while communicating to the peer an unsigned 32-bit error code and an error reason string of at most 1024 bytes. The error code and string are optional; the default values are 0 and "".

Any WebTransport protocol SHALL provide the following events:

session terminated event Indicates that the WebTransport session has been terminated, either by the peer or by the local networking stack, and no user data can be exchanged on it any further. If the session has been terminated as a result of the peer performing the "terminate a session" operation above, a corresponding error code and an error string can be provided.

4.2. Datagrams

A datagram is a sequence of bytes that is limited in size (generally to the path MTU) and is not expected to be transmitted reliably. The general goal for WebTransport datagrams is to be similar in behavior to UDP while being subject to common requirements expressed in Section 2.

A WebTransport sender is not expected to retransmit datagrams, though it may end up doing so if it is using a TCP-based protocol or some other underlying protocol that only provides reliable delivery. WebTransport datagrams are not expected to be flow controlled,

meaning that the receiver might drop datagrams if the application is not consuming them fast enough.

The application MUST be provided with the maximum datagram size that it can send. The size SHOULD be derived from the result of performing path MTU discovery.

In the WebTransport model, all of the outgoing and incoming datagrams are placed into a size-bound queue (similar to a network interface card queue).

Any WebTransport protocol SHALL provide the following operations on the session:

send a datagram Enqueues a datagram to be sent to the peer. This can potentially result in the datagram being dropped if the queue is full.

receive a datagram Dequeues an incoming datagram, if one is available.

get maxiumum datagram size Returns the largest size of the datagram that a WebTransport session is expected to be able to send.

4.3. Streams

A unidirectional stream is a one-way reliable in-order stream of bytes where the initiator is the only endpoint that can send data. A bidirectional stream allows both endpoints to send data and can be conceptually represented as a pair of unidirectional streams.

The streams are in general expected to follow the semantics and the state machine of QUIC streams ([RFC9000], Sections 2 and 3). TODO: describe the stream state machine explicitly.

A WebTransport stream can be reset, indicating that the endpoint is not interested in either sending or receiving any data related to the stream. In that case, the sender is expected to not retransmit any data that was already sent on that stream.

Streams SHOULD be sufficiently lightweight that they can be used as messages.

Data sent on a stream is flow controlled by the transport protocol. In addition to flow controlling stream data, the creation of new streams is flow controlled as well: an endpoint may only open a limited number of streams until the peer explicitly allows creating more streams. From the perspective of the client, this is presented as a size-bounded queue of incoming streams.

Any WebTransport protocol SHALL provide the following operations on the session:

create a unidirectional stream Creates an outgoing unidirectional stream; this operation may block until the flow control of the underlying protocol allows for it to be completed.

create a bidirectional stream Creates an outgoing bidirectional stream; this operation may block until the flow control of the underlying protocol allows for it to be completed.

receive a unidirectional stream Removes a stream from the queue of incoming unidirectional streams, if one is available.

receive a bidirectional stream Removes a stream from the queue of incoming unidirectional streams, if one is available.

Any WebTransport protocol SHALL provide the following operations on an individual stream:

send bytes Add bytes into the stream send buffer. The sender can also indicate a FIN, signalling the fact that no new data will be send on the stream. Not applicable for incoming unidirectional streams.

receive bytes Removes bytes from the stream receive buffer. FIN can be received together with the stream data. Not applicable for outgoing unidirectional streams.

abort send side Sends a signal to the peer that the write side of the stream has been aborted. Discards the send buffer; if possible, no currently outstanding data is transmitted or retransmitted. An unsigned 8-bit error code can be supplied as a part of the signal to the peer; if omitted, the error code is presumed to be 0.

abort receive side Sends a signal to the peer that the read side of the stream has been aborted. Discards the receive buffer; the peer is typically expected to abort the corresponding send side in response. An unsigned 8-bit error code can be supplied as a part of the signal to the peer.

Any WebTransport protocol SHALL provide the following events for an individual stream:

send side aborted
: Indicates that the peer has aborted the corresponding receive side of the stream. An unsigned 8-bit error code from the peer may be available.

receive side aborted
: Indicates that the peer has aborted the corresponding send side of the stream. An unsigned 8-bit error code from the peer may be available.

Data Recvd state reached
: Indicates that no further data will be transmitted or retransmitted on the local send side, and that the FIN has been sent. Data Recvd implies that aborting send-side is a no-op.

5. Transport Properties

WebTransport defines common semantics for multiple protocols to allow them to be used interchangeably. Nevertheless, those protocols still have substantially different performance properties that an application may want to query.

The most notable property is support for unreliable data delivery. The protocol is defined to support unreliable delivery if:

* Resetting a stream results in the lost stream data no longer being retransmitted, and

* The datagrams are never retransmitted.

Another important property is pooling support. Pooling means that multiple transport sessions may end up sharing the same transport layer connection, and thus share a congestion controller and other contexts.

6. Security Considerations

Providing untrusted clients with a reasonably low-level access to the network comes with risks. This document mitigates those risks by imposing a set of common requirements described in Section 2.

WebTransport mandates the use of TLS for all protocols implementing it. This has a dual purpose. On one hand, it protects the transport from the network, including both potential attackers and ossification by middleboxes. On the other hand, it protects the network elements from potential confusion attacks such as the one discussed in Section 10.3 of [RFC6455].

One potential concern is that even when a transport cannot be created, the connection error would reveal enough information to allow an attacker to scan the network addresses that would normally

be inaccessible. Because of that, the user agent that runs untrusted clients MUST NOT provide any detailed error information until the server has confirmed that it is a WebTransport endpoint. For example, the client must not be able to distinguish between a network address that is unreachable and one that is reachable but is not a WebTransport server.

WebTransport does not support any traditional means of HTTP-based authentication. It is not necessarily based on HTTP, and hence does not support HTTP cookies or HTTP authentication. Since it requires TLS, individual transport protocols MAY expose TLS-based authentication capabilities such as client certificates.

7. IANA Considerations

There are no requests to IANA in this document.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/rfc/rfc6454>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

8.2. Informative References

[CSP] W3C, "Content Security Policy Level 3", July 2022, <https://www.w3.org/TR/CSP/>.

[I-D.ietf-httpbis-h3-websockets] Hamilton, R., "Bootstrapping WebSockets with HTTP/3", Work in Progress, Internet-Draft, draft-ietf-httpbis-h3-websockets-04, 8 February 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-h3-websockets-04>.

[I-D.ietf-quic-datagram] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-datagram-10, 4 February 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-10>.

[I-D.ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-43>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/rfc/rfc5681>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/rfc/rfc6455>.

[RFC7675] Perumal, M., Wing, D., Ravindranath, R., Reddy, T., and M. Thomson, "Session Traversal Utilities for NAT (STUN) Usage for Consent Freshness", RFC 7675, DOI 10.17487/RFC7675, October 2015, <https://www.rfc-editor.org/rfc/rfc7675>.

[RFC8441] McManus, P., "Bootstrapping WebSockets with HTTP/2", RFC 8441, DOI 10.17487/RFC8441, September 2018, <https://www.rfc-editor.org/rfc/rfc8441>.

[RFC8831] Jesup, R., Loreto, S., and M. Tüxen, "WebRTC Data Channels", RFC 8831, DOI 10.17487/RFC8831, January 2021, <https://www.rfc-editor.org/rfc/rfc8831>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9002]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/rfc/rfc9002>.

Author's Address

Victor Vasiliev
Google

Email: vasilvv@google.com