W3C Editor's Draft

WebTransport

Editor's Draft, 28 May 2025

▼ **More details about this document**

This version:
https://w3c.github.io/webtransport/

Latest published version:
https://www.w3.org/TR/webtransport/

Feedback:
public-webtransport@w3.org with subject line "`[webtransport] … message topic …`" (archives)
GitHub
Inline In Spec

Editors:
Nidhi Jaju (Google)
Victor Vasiliev (Google)
Jan-Ivar Bruaroey (Mozilla)

Former Editors:
Bernard Aboba (Microsoft Corporation)
Peter Thatcher (Google)
Robin Raymond (Optical Tone Ltd.)
Yutaka Hirano (Google)

Abstract

This document defines a set of ECMAScript APIs in WebIDL to allow data to be sent and received between a browser and server, utilizing [WEB-TRANSPORT-HTTP3] and [WEB-TRANSPORT-HTTP2]. This specification is being developed in conjunction with protocol specifications developed by the IETF WEBTRANS Working Group.

Status of this document

This document is governed by the 03 November 2023 W3C Process Document.

Table of Contents

§ 1. Introduction

This section is non-normative.

This specification uses [WEB-TRANSPORT-HTTP3] and [WEB-TRANSPORT-HTTP2] to send data to and receive data from servers. It can be used like WebSockets but with support for multiple streams, unidirectional streams, out-of-order delivery, and reliable as well as unreliable transport.

NOTE: The API presented in this specification represents a preliminary proposal based on work-in-progress within the IETF WEBTRANS WG. Since the [WEB-TRANSPORT-HTTP3] and [WEB-TRANSPORT-HTTP2] specifications are a work-in-progress, both the protocol and API are likely to change significantly going forward.

§ 2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

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

This specification defines conformance criteria that apply to a single product: the user agent that implements the interfaces that it contains.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [WEBIDL], as this specification uses that specification and terminology.

§ 3. Protocol concepts

There are two main protocol concepts for WebTransport: sessions and streams. Each WebTransport session can contain multiple WebTransport streams.

These should not be confused with protocol names which is an application-level API construct.

§ 3.1. WebTransport session

A ***WebTransport session*** is a session of WebTransport over an HTTP/3 or HTTP/2 ***underlying connection***. There may be multiple WebTransport sessions on one connection, when pooling is enabled.

A WebTransport session has the following capabilities defined in [WEB-TRANSPORT-OVERVIEW]:

capability	definition
send a datagram	[WEB-TRANSPORT-OVERVIEW] Section 4.2
receive a datagram	[WEB-TRANSPORT-OVERVIEW] Section 4.2
create an outgoing unidirectional stream	[WEB-TRANSPORT-OVERVIEW] Section 4.3
create a bidirectional stream	[WEB-TRANSPORT-OVERVIEW] Section 4.3
receive an incoming unidirectional stream	[WEB-TRANSPORT-OVERVIEW] Section 4.3
receive a bidirectional stream	[WEB-TRANSPORT-OVERVIEW] Section 4.3

To ***establish*** a WebTransport session with an origin *origin* and a *protocols* array, follow [WEB-TRANSPORT-OVERVIEW] Section 4.1, using *origin*, serialized and isomorphically encoded, as the `Origin` header of the request, and the isomorphically encoded *protocols* as the list of protocols the client would like the server to use in this session, in preference order, following [WEB-TRANSPORT-OVERVIEW] Section 3.1. When establishing a session, the client MUST NOT provide any credentials. The resulting underlying transport stream is referred to as the session's ***CONNECT stream***.

A WebTransport session *session* is ***draining*** when the CONNECT stream receives a WT_DRAIN_SESSION capsule, or when a GOAWAY frame is received, as described in [WEB-TRANSPORT-OVERVIEW] Section 4.1.

To ***terminate*** a WebTransport session *session* with an optional integer *code* and an optional byte sequence *reason*, follow [WEB-TRANSPORT-OVERVIEW] Section 4.1.

A WebTransport session *session* is ***terminated***, with optionally an integer *code* and a byte sequence *reason*, when the CONNECT stream is closed by the server, as described at [WEB-TRANSPORT-OVERVIEW] Section 4.1.

A WebTransport session has the following signals:

event	definition
WT_DRAIN_SESSION	[WEB-TRANSPORT-OVERVIEW] Section 4.1
GOAWAY	[WEB-TRANSPORT-OVERVIEW] Section 4.1

§ 3.2. WebTransport stream

A ***WebTransport stream*** is a concept for a reliable in-order stream of bytes on a WebTransport session, as described in [WEB-TRANSPORT-OVERVIEW] Section 4.3.

A WebTransport stream is one of ***incoming unidirectional***, ***outgoing unidirectional*** or ***bidirectional***.

A WebTransport stream has the following capabilities:

capability	definition	**incoming unidirectional**	**outgoing unidirectional**	**bidirectional**
send bytes (potentially with FIN)	[WEB-TRANSPORT-OVERVIEW] Section 4.3	No	Yes	Yes
receive bytes (potentially with FIN)	[WEB-TRANSPORT-OVERVIEW] Section 4.3	Yes	No	Yes
send STOP_SENDING	[WEB-TRANSPORT-OVERVIEW] Section 4.3	Yes	No	Yes
reset a WebTransport stream	[WEB-TRANSPORT-OVERVIEW] Section 4.3	No	Yes	Yes

A WebTransport stream has the following signals:

event	definition	**incoming unidirectional**	**outgoing unidirectional**	**bidirectional**
STOP_SENDING	[WEB-TRANSPORT-OVERVIEW] Section 4.3	No	Yes	Yes
RESET_STREAM	[WEB-TRANSPORT-OVERVIEW] Section 4.3	Yes	No	Yes

event	definition	incoming unidirectional	outgoing unidirectional	bidirectional
flow control	[WEB-TRANSPORT-OVERVIEW] Section 4.3	No	Yes	Yes

§ 4. `WebTransportDatagramsWritable` Interface

A ***`WebTransportDatagramsWritable`*** is a `WritableStream` providing outgoing streaming features to send datagrams.

```
[Exposed=(Window,Worker), SecureContext, Transferable]
interface WebTransportDatagramsWritable : WritableStream {
  attribute WebTransportSendGroup? sendGroup;
  attribute long long sendOrder;
};
```

§ 4.1. Internal slots

A `WebTransportDatagramsWritable` object has the following internal slot.

Internal Slot	Description (*non-normative*)
`[[OutgoingDatagramsQueue]]`	A queue of tuples of an outgoing datagram, a timestamp and a promise which is resolved when the datagram is sent or discarded.
`[[Transport]]`	A `WebTransport` which owns this `WebTransportDatagramsWritable`.
`[[SendGroup]]`	An optional `WebTransportSendGroup`, or null.
`[[SendOrder]]`	An optional send order number, defaulting to 0.

To ***create*** a `WebTransportDatagramsWritable`, given a `WebTransport` *transport*, a *sendGroup*, and a *sendOrder*, perform the following steps.

1. Let *stream* be a new `WebTransportDatagramsWritable`, with:

   `[[OutgoingDatagramsQueue]]`
   : an empty queue

   `[[Transport]]`
   : *transport*

   `[[SendGroup]]`
   : *sendGroup*

[[SendOrder]]
: *sendOrder*

2. Let *writeDatagramsAlgorithm* be an action that runs writeDatagrams with *transport* and *stream*.
3. Set up *stream* with writeAlgorithm set to *writeDatagramsAlgorithm*.
4. Return *stream*.

§ 4.2. Attributes

***sendGroup*, of type WebTransportSendGroup, nullable**

The getter steps are:

1. Return this's `[[SendGroup]]`.

The setter steps, given *value*, are:

1. If *value* is non-null, and *value*.`[[Transport]]` is not this.`[[Transport]]`, throw an `InvalidStateError`.
2. Set this.`[[SendGroup]]` to *value*.

***sendOrder*, of type long long**

The getter steps are:

1. Return this's `[[SendOrder]]`.

The setter steps, given *value*, are:

1. Set this.`[[SendOrder]]` to *value*.

§ 4.3. Procedures

The ***writeDatagrams*** algorithm is given a *transport* and *writable* as parameters and *data* as input. It is defined by running the following steps:

1. Let *timestamp* be a timestamp representing now.
2. If *data* is not a `BufferSource` object, then return a promise rejected with a `TypeError`.
3. Let *datagrams* be *transport*.`[[Datagrams]]`.
4. If *datagrams*.`[[OutgoingMaxDatagramSize]]` is less than *data*'s [[ByteLength]], return a promise resolved with undefined.
5. Let *promise* be a new promise.
6. Let *bytes* be a copy of bytes which *data* represents.
7. Let *chunk* be a tuple of *bytes*, *timestamp* and *promise*.
8. Enqueue *chunk* to *writable*.`[[OutgoingDatagramsQueue]]`.
9. If the length of *writable*.`[[OutgoingDatagramsQueue]]` is less than *datagrams*.`[[OutgoingDatagramsHighWaterMark]]`, then resolve *promise* with undefined.
10. Return *promise*.

NOTE: The associated `WritableStream` calls writeDatagrams only when all the promises that have been returned by writeDatagrams for that stream have been resolved. Hence the timestamp and the expiration duration work well only when the web developer pays attention to `WritableStreamDefaultWriter.ready`.

To ***sendDatagrams***, given a `WebTransport` object *transport* and a `WebTransportDatagramsWritable` object *writable*, run these steps:

1. Let *queue* be *writable*.`[[OutgoingDatagramsQueue]]`.
2. Let *datagrams* be *transport*.`[[Datagrams]]`.
3. Let *duration* be *datagrams*.`[[OutgoingDatagramsExpirationDuration]]`.
4. If *duration* is null, then set *duration* to an implementation-defined value.
5. While *queue* is not empty:
    1. Let *bytes*, *timestamp* and *promise* be *queue*'s first element.
    2. If more than *duration* milliseconds have passed since *timestamp*, then:
        1. Remove the first element from *queue*.
        2. Queue a network task with *transport* to resolve *promise* with undefined.
    3. Otherwise, break this loop.
6. If *transport*.`[[State]]` is not `"connected"`, then return.
7. Let *maxSize* be *datagrams*.`[[OutgoingMaxDatagramSize]]`.
8. While *queue* is not empty:
    1. Let *bytes*, *timestamp* and *promise* be *queue*'s first element.
    2. If *bytes*'s length ≤ *maxSize*:
        1. If it is not possible to send *bytes* to the network immediately, then break this loop.
        2. Send a datagram, with *transport*.`[[Session]]` and *bytes*.
    3. Remove the first element from *queue*.
    4. Queue a network task with *transport* to resolve *promise* with undefined.

The user agent SHOULD run sendDatagrams for any `WebTransport` object whose `[[State]]` is `"connecting"` or `"connected"` as soon as reasonably possible on a subset, based on send-order rules, of its associated `WebTransportDatagramsWritable` objects whenever the algorithm can make progress.

The ***send-order rules*** are that sending in general MAY be interleaved with sending of previously queued streams and datagrams, as well as streams and datagrams yet to be queued to be sent over this transport, except that sending MUST starve until all bytes queued for sending on streams and datagrams with the same `[[SendGroup]]` and a higher `[[SendOrder]]`, that are neither errored nor blocked by flow control, have been sent.

NOTE: Writing datagrams while the transport's `[[State]]` is `"connecting"` is allowed. The datagrams are stored in `[[OutgoingDatagramsQueue]]`, and they can be discarded in the same manner as when in the `"connected"` state. Once the transport's `[[State]]` becomes `"connected"`, it will start sending the queued datagrams.

§ 5. `WebTransportDatagramDuplexStream` Interface

A ***`WebTransportDatagramDuplexStream`*** is a generic duplex stream.

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransportDatagramDuplexStream {
  WebTransportDatagramsWritable createWritable(
      optional WebTransportSendOptions options = {});
  readonly attribute ReadableStream readable;

  readonly attribute unsigned long maxDatagramSize;
  attribute unrestricted double? incomingMaxAge;
  attribute unrestricted double? outgoingMaxAge;
  attribute unrestricted double incomingHighWaterMark;
  attribute unrestricted double outgoingHighWaterMark;
};
```

§ 5.1. Internal slots

A `WebTransportDatagramDuplexStream` object has the following internal slots.

Internal Slot	**Description (*non-normative*)**
`[[Readable]]`	A `ReadableStream` for incoming datagrams.
`[[Writables]]`	An ordered set of `WebTransportDatagramsWritable` streams, initially empty.
`[[IncomingDatagramsQueue]]`	A queue of pairs of an incoming datagram and a timestamp.
`[[IncomingDatagramsPullPromise]]`	A promise set by pullDatagrams, to wait for an incoming datagram.
`[[IncomingDatagramsHighWaterMark]]`	An `unrestricted double` representing the high water mark of the incoming datagrams.
`[[IncomingDatagramsExpirationDuration]]`	An `unrestricted double` representing the expiration duration for incoming datagrams (in milliseconds), or null.
`[[OutgoingDatagramsHighWaterMark]]`	An `unrestricted double` representing the high water mark of the outgoing datagrams.
`[[OutgoingDatagramsExpirationDuration]]`	An `unrestricted double` value representing the expiration duration for outgoing datagrams (in milliseconds), or null.
`[[OutgoingMaxDatagramSize]]`	An integer representing the maximum size for an outgoing datagram.

The user agent MAY update `[[OutgoingMaxDatagramSize]]` for any `WebTransport` object whose `[[State]]` is either `"connecting"` or `"connected"`.

To ***create*** a `WebTransportDatagramDuplexStream` given a **readable**, perform the following steps.

1. Let *stream* be a new `WebTransportDatagramDuplexStream`, with:

   `[[Readable]]`
   : *readable*

   `[[Writables]]`
   : an empty ordered set.

   `[[IncomingDatagramsQueue]]`
   : an empty queue

   `[[IncomingDatagramsPullPromise]]`
   : null

   `[[IncomingDatagramsHighWaterMark]]`
   : an implementation-defined value

   `[[IncomingDatagramsExpirationDuration]]`
   : null

   `[[OutgoingDatagramsHighWaterMark]]`
   : an implementation-defined value

   > This implementation-defined value should be tuned to ensure decent throughput, without jeopardizing the timeliness of transmitted data.

   `[[OutgoingDatagramsExpirationDuration]]`
   : null

   `[[OutgoingMaxDatagramSize]]`
   : an implementation-defined integer.

2. Return *stream*.

§ 5.2. Methods

`createWritable()`

Creates a `WebTransportDatagramsWritable`.

When `createWritable()` method is called, the user agent MUST run the following steps:

1. Let *transport* be `WebTransport` object associated with this.
2. If *transport*.`[[State]]` is `"closed"` or `"failed"`, throw an `InvalidStateError`.
3. Let *sendGroup* be `options`'s `sendGroup`.
4. Let *sendOrder* be `options`'s `sendOrder`.
5. Return the result of creating a `WebTransportDatagramsWritable` with *transport*, *sendGroup* and *sendOrder*.

§ 5.3. Attributes

readable, **of type ReadableStream, readonly**

The getter steps are:

1. Return this.`[[Readable]]`.

incomingMaxAge, **of type unrestricted double, nullable**

The getter steps are:

1. Return this.`[[IncomingDatagramsExpirationDuration]]`.

The setter steps, given *value*, are:

1. If *value* is negative or NaN, throw a `RangeError`.
2. If *value* is `0`, set *value* to null.
3. Set this.`[[IncomingDatagramsExpirationDuration]]` to *value*.

maxDatagramSize, **of type unsigned long, readonly**

The maximum size data that may be passed to a `WebTransportDatagramsWritable`. The getter steps are to return this.`[[OutgoingMaxDatagramSize]]`.

outgoingMaxAge, **of type unrestricted double, nullable**

The getter steps are:

1. Return this's `[[OutgoingDatagramsExpirationDuration]]`.

The setter steps, given *value*, are:

1. If *value* is negative or NaN, throw a `RangeError`.
2. If *value* is `0`, set *value* to null.
3. Set this.`[[OutgoingDatagramsExpirationDuration]]` to *value*.

incomingHighWaterMark, **of type unrestricted double**

The getter steps are:

1. Return this.`[[IncomingDatagramsHighWaterMark]]`.

The setter steps, given *value*, are:

1. If *value* is negative or NaN, throw a `RangeError`.
2. If *value* is < `1`, set *value* to `1`.
3. Set this.`[[IncomingDatagramsHighWaterMark]]` to *value*.

outgoingHighWaterMark, **of type unrestricted double**

The getter steps are:

1. Return this.`[[OutgoingDatagramsHighWaterMark]]`.

The setter steps, given *value*, are:

1. If *value* is negative or NaN, throw a `RangeError`.
2. If *value* is < `1`, set *value* to `1`.
3. Set this.`[[OutgoingDatagramsHighWaterMark]]` to *value*.

§ 5.4. Procedures

To ***pullDatagrams***, given a `WebTransport` object *transport*, run these steps:

1. Let *datagrams* be *transport*.`[[Datagrams]]`.
2. Assert: *datagrams*.`[[IncomingDatagramsPullPromise]]` is null.
3. Let *queue* be *datagrams*.`[[IncomingDatagramsQueue]]`.
4. If *queue* is empty, then:
    1. Set *datagrams*.`[[IncomingDatagramsPullPromise]]` to a new promise.
    2. Return *datagrams*.`[[IncomingDatagramsPullPromise]]`.
5. Let *datagram* and *timestamp* be the result of dequeuing *queue*.
6. If *datagrams*.`[[Readable]]`’s current BYOB request view is not null, then:
    1. Let *view* be *datagrams*.`[[Readable]]`’s current BYOB request view.
    2. If *view*’s byte length is less than the size of *datagram*, return a promise rejected with a `RangeError`.
    3. Let *elementSize* be the element size specified in the typed array constructors table for *view*.[[TypedArrayName]]. If *view* does not have a [[TypedArrayName]] internal slot (i.e. it is a `DataView`), let *elementSize* be 0.
    4. If *elementSize* is not 1, return a promise rejected with a `TypeError`.
7. Pull from bytes *datagram* into *datagrams*.`[[Readable]]`.
8. Return a promise resolved with undefined.

To ***receiveDatagrams***, given a `WebTransport` object *transport*, run these steps:

1. Let *timestamp* be a timestamp representing now.
2. Let *queue* be *datagrams*.`[[IncomingDatagramsQueue]]`.
3. Let *duration* be *datagrams*.`[[IncomingDatagramsExpirationDuration]]`.
4. If *duration* is null, then set *duration* to an implementation-defined value.
5. Let *session* be *transport*.`[[Session]]`.
6. While there are available incoming datagrams on *session*:
    1. Let *datagram* be the result of receiving a datagram with *session*.
    2. Let *timestamp* be a timestamp representing now.
    3. Let *chunk* be a pair of *datagram* and *timestamp*.
    4. Enqueue *chunk* to *queue*.
7. Let *toBeRemoved* be the length of *queue* minus *datagrams*.`[[IncomingDatagramsHighWaterMark]]`.
8. If *toBeRemoved* is positive, repeat dequeuing *queue* *toBeRemoved* (rounded down) times.
9. While *queue* is not empty:
    1. Let *bytes* and *timestamp* be *queue*’s first element.
    2. If more than *duration* milliseconds have passed since *timestamp*, then dequeue *queue*.
    3. Otherwise, break this loop.

10. If *queue* is not empty and *datagrams*.[[IncomingDatagramsPullPromise]] is non-null, then:
    1. Let *bytes* and *timestamp* be the result of dequeuing *queue*.
    2. Let *promise* be *datagrams*.[[IncomingDatagramsPullPromise]].
    3. Set *datagrams*.[[IncomingDatagramsPullPromise]] to null.
    4. Queue a network task with *transport* to run the following steps:
        1. Let *chunk* be a new Uint8Array object representing *bytes*.
        2. Enqueue *chunk* to *datagrams*.[[Readable]].
        3. Resolve *promise* with undefined.

The user agent SHOULD run receiveDatagrams for any WebTransport object whose [[State]] is "connected" as soon as reasonably possible whenever the algorithm can make progress.

§ 6. WebTransport Interface

WebTransport provides an API to the underlying transport functionality defined in [WEB-TRANSPORT-OVERVIEW].

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransport {
  constructor(USVString url, optional WebTransportOptions options = {});

  Promise<WebTransportConnectionStats> getStats();
  [NewObject] Promise<ArrayBuffer> exportKeyingMaterial(BufferSource label, optional Buffe
  readonly attribute Promise<undefined> ready;
  readonly attribute WebTransportReliabilityMode reliability;
  readonly attribute WebTransportCongestionControl congestionControl;
  [EnforceRange] attribute unsigned short? anticipatedConcurrentIncomingUnidirectionalStre
  [EnforceRange] attribute unsigned short? anticipatedConcurrentIncomingBidirectionalStrea
  readonly attribute DOMString protocol;

  readonly attribute Promise<WebTransportCloseInfo> closed;
  readonly attribute Promise<undefined> draining;
  undefined close(optional WebTransportCloseInfo closeInfo = {});

  readonly attribute WebTransportDatagramDuplexStream datagrams;

  Promise<WebTransportBidirectionalStream> createBidirectionalStream(
      optional WebTransportSendStreamOptions options = {});
  /* a ReadableStream of WebTransportBidirectionalStream objects */
  readonly attribute ReadableStream incomingBidirectionalStreams;

  Promise<WebTransportSendStream> createUnidirectionalStream(
      optional WebTransportSendStreamOptions options = {});
  /* a ReadableStream of WebTransportReceiveStream objects */
  readonly attribute ReadableStream incomingUnidirectionalStreams;
  WebTransportSendGroup createSendGroup();

  static readonly attribute boolean supportsReliableOnly;
};

enum WebTransportReliabilityMode {
  "pending",
  "reliable-only",
  "supports-unreliable",
};
```

§ 6.1. Internal slots

A `WebTransport` object has the following internal slots.

Internal Slot	Description (*non-normative*)
[[SendStreams]]	An ordered set of `WebTransportSendStream`s owned by this `WebTransport`.
[[ReceiveStreams]]	An ordered set of `WebTransportReceiveStream`s owned by this `WebTransport`.
[[IncomingBidirectionalStreams]]	A `ReadableStream` consisting of `WebTransportBidirectionalStream` objects.
[[IncomingUnidirectionalStreams]]	A `ReadableStream` consisting of `WebTransportReceiveStream`s.
[[State]]	An enum indicating the state of the transport. One of `"connecting"`, `"connected"`, `"draining"`, `"closed"`, and `"failed"`.
[[Ready]]	A promise fulfilled when the associated WebTransport session gets established, or rejected if the establishment process failed.
[[Reliability]]	A `WebTransportReliabilityMode` indicating whether the first hop supports unreliable (UDP) transport or whether only reliable (TCP fallback) transport is available. Returns `"pending"` until a connection has been established.
[[CongestionControl]]	A `WebTransportCongestionControl` indicating whether a preference for a congestion control algorithm optimized for throughput or

Internal Slot	Description (*non-normative*)
	low latency was requested by the application and satisfied by the user agent, or `"default"`.
`[[AnticipatedConcurrentIncomingUnidirectionalStreams]]`	The number of concurrently open incoming unidirectional streams the application anticipates the server creating, or null.
`[[AnticipatedConcurrentIncomingBidirectionalStreams]]`	The number of concurrently open bidirectional streams the application anticipates the server creating, or null.
`[[Protocol]]`	A string indicating the application-level protocol selected by the server, if any. Initially an empty string.
`[[Closed]]`	A promise fulfilled when the associated `WebTransport` object is closed gracefully, or rejected when it is closed abruptly or failed on initialization.
`[[Draining]]`	A promise fulfilled when the associated WebTransport session receives a WT_DRAIN_SESSION capsule or a GOAWAY frame.
`[[Datagrams]]`	A `WebTransportDatagramDuplexStream`.
`[[Session]]`	A WebTransport session for this `WebTransport` object, or null.

§ 6.2. Constructor

When the `WebTransport()` constructor is invoked, the user agent MUST run the following steps:

1. Let *baseURL* be this's relevant settings object's API base URL.
2. Let *parsedURL* be the URL record resulting from parsing `url` with *baseURL*.
3. If *parsedURL* is a failure, throw a `SyntaxError` exception.
4. If *parsedURL* scheme is not `https`, throw a `SyntaxError` exception.
5. If *parsedURL* fragment is not null, throw a `SyntaxError` exception.
6. Let *allowPooling* be `options`'s `allowPooling`.
7. Let *dedicated* be the negation of *allowPooling*.
8. Let *serverCertificateHashes* be `options`'s `serverCertificateHashes` if it exists, and null otherwise.
9. If *dedicated* is false and *serverCertificateHashes* is non-null, then throw a `NotSupportedError` exception.

10. Let *requireUnreliable* be `options`’s `requireUnreliable`.
11. Let *congestionControl* be `options`’s `congestionControl`.
12. If *congestionControl* is not `"default"`, and the user agent does not support any congestion control algorithms that optimize for *congestionControl*, as allowed by [RFC9002] Section 7, then set *congestionControl* to `"default"`.
13. Let *protocols* be `options`’s `protocols`
14. If any of the values in *protocols* occur more than once, fail to match the requirements for elements that comprise the value of the negotiated application protocol as defined by the WebTransport protocol, or have an isomorphic encoded length of 0 or exceeding 512, throw a `SyntaxError` exception. [WEB-TRANSPORT-OVERVIEW] Section 3.1.
15. Let *anticipatedConcurrentIncomingUnidirectionalStreams* be `options`’s `anticipatedConcurrentIncomingUnidirectionalStreams`.
16. Let *anticipatedConcurrentIncomingBidirectionalStreams* be `options`’s `anticipatedConcurrentIncomingBidirectionalStreams`.
17. Let *incomingDatagrams* be a new `ReadableStream`.
18. Let *datagrams* be the result of creating a `WebTransportDatagramDuplexStream`, its readable set to *incomingDatagrams*.
19. Let *transport* be a newly constructed `WebTransport` object, with:

    `[[SendStreams]]`
    : an empty ordered set

    `[[ReceiveStreams]]`
    : an empty ordered set

    `[[IncomingBidirectionalStreams]]`
    : a new `ReadableStream`

    `[[IncomingUnidirectionalStreams]]`
    : a new `ReadableStream`

    `[[State]]`
    : `"connecting"`

    `[[Ready]]`
    : a new promise

    `[[Reliability]]`
    : "pending"

    `[[CongestionControl]]`
    : *congestionControl*

    `[[AnticipatedConcurrentIncomingUnidirectionalStreams]]`
    : *anticipatedConcurrentIncomingUnidirectionalStreams*

    `[[AnticipatedConcurrentIncomingBidirectionalStreams]]`
    : *anticipatedConcurrentIncomingBidirectionalStreams*

    `[[Protocol]]`
    : an empty string

    `[[Closed]]`
    : a new promise

    `[[Draining]]`
    : a new promise

`[[Datagrams]]`
: *datagrams*

`[[Session]]`
: null

20. Let *pullDatagramsAlgorithm* be an action that runs pullDatagrams with *transport*.

NOTE: Using 64kB buffers with datagrams is recommended because the effective maximum WebTransport datagram frame size has an upper bound of the QUIC maximum datagram frame size which is recommended to be 64kB (See [QUIC-DATAGRAM] Section 3). This will ensure the stream is not errored due to a datagram being larger than the buffer.

1. Set up with byte reading support *incomingDatagrams* with pullAlgorithm set to *pullDatagramsAlgorithm*, and highWaterMark set to 0.
2. Let *pullBidirectionalStreamAlgorithm* be an action that runs pullBidirectionalStream with *transport*.
3. Set up *transport*.`[[IncomingBidirectionalStreams]]` with pullAlgorithm set to *pullBidirectionalStreamAlgorithm*, and highWaterMark set to 0.
4. Let *pullUnidirectionalStreamAlgorithm* be an action that runs pullUnidirectionalStream with *transport*.
5. Set up *transport*.`[[IncomingUnidirectionalStreams]]` with pullAlgorithm set to *pullUnidirectionalStreamAlgorithm*, and highWaterMark set to 0.
6. Initialize WebTransport over HTTP with *transport*, *parsedURL*, *dedicated*, *requireUnreliable*, *congestionControl*, *protocols*, and *serverCertificateHashes*.
7. Return *transport*.

To ***initialize WebTransport over HTTP***, given a `WebTransport` object *transport*, a URL record *url*, a boolean *dedicated*, a boolean *requireUnreliable*, a `WebTransportCongestionControl` *congestionControl*, a *protocols* array, and a sequence<`WebTransportHash`> *serverCertificateHashes*, run these steps.

1. Let *client* be *transport*'s relevant settings object.
2. Let *origin* be *client*'s origin.
3. Let *request* be a new request whose URL is *url*, client is *client*, policy container is *client*'s policy container, destination is an empty string, origin is *origin* and redirect mode is "error".
4. Run report Content Security Policy violations for *request*.
5. If should request be blocked by Content Security Policy? with *request* returns **"Blocked"**, or if *request* should be blocked due to a bad port returns **blocked**, then abort the remaining steps and queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, then abort these steps.
    2. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    3. Cleanup *transport* with *error*.
6. Let *networkPartitionKey* be the result of determining the network partition key with *transport*'s relevant settings object.
7. Run the following steps in parallel, but abort when *transport*.`[[State]]` becomes `"closed"` or `"failed"`:
    1. Let *newConnection* be "`no`" if *dedicated* is false; otherwise "`yes-and-dedicated`".

2. Let *connection* be the result of obtaining a connection with *networkPartitionKey*, *url*, false, *newConnection*, and *requireUnreliable*. If the user agent supports more than one congestion control algorithm, choose one appropriate for *congestionControl* for sending of data on this *connection*. When obtaining a connection, if *serverCertificateHashes* is specified, instead of using the default certificate verification algorithm, consider the certificate valid if it meets the custom certificate requirements and if verifying the certificate hash against *serverCertificateHashes* returns true. If either condition is not met, let *connection* be failure.
3. If *connection* is failure, then abort the remaining steps and queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, then abort these steps.
    2. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    3. Cleanup *transport* with *error*.

    NOTE: Redirects are not followed. Network errors caused by redirection are intentionally indistinguishable from other network errors. In cross-origin contexts, this would reveal information that would normally be blocked by CORS. In same-origin contexts, it might encourage applications to abuse the handshake as a vector for passing information.
4. Wait for *connection* to receive the first SETTINGS frame, and let *settings* be a dictionary that represents the SETTINGS frame.
5. If *settings* doesn’t contain SETTINGS_ENABLE_WEBTRANPORT with a value of 1, or it doesn’t contain H3_DATAGRAM with a value of 1, then abort the remaining steps and queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, then abort these steps.
    2. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    3. Cleanup *transport* with *error*.
6. Establish a WebTransport session with *origin* and *protocols* on *connection*.

    NOTE: This step also contains the transport parameter exchange specified in [QUIC-DATAGRAM].
7. If the previous step fails, abort the remaining steps and queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, then abort these steps.
    2. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    3. Cleanup *transport* with *error*.
8. Let *session* be the established WebTransport session.
9. Assert: *maxDatagramSize* is an integer.
10. Queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is not `"connecting"`:
        1. In parallel, terminate *session*.
        2. Abort these steps.
    2. Set *transport*.`[[State]]` to `"connected"`.
    3. Set *transport*.`[[Session]]` to *session*.

        4. Set *transport*.`[[Protocol]]` to either the string value of the negotiated application protocol if present, following [WEB-TRANSPORT-OVERVIEW] Section 3.1, or `""` if not present.
        5. If the connection is an HTTP/3 connection, set *transport*.`[[Reliability]]` to `"supports-unreliable"`.
        6. If the connection is an HTTP/2 connection [WEB-TRANSPORT-HTTP2], set *transport*'s `[[Reliability]]` to `"reliable-only"`.
        7. Resolve *transport*.`[[Ready]]` with undefined.

To ***pullBidirectionalStream***, given a `WebTransport` object *transport*, run these steps.

1. If *transport*.`[[State]]` is `"connecting"`, then return the result of performing the following steps upon fulfillment of *transport*.`[[Ready]]`:
    1. Return the result of pullBidirectionalStream with *transport*.
2. If *transport*.`[[State]]` is not `"connected"`, then return a new rejected promise with an `InvalidStateError`.
3. Let *session* be *transport*.`[[Session]]`.
4. Let *p* be a new promise.
5. Run the following steps in parallel:
    1. Wait until there is an available incoming bidirectional stream.
    2. Let *internalStream* be the result of receiving a bidirectional stream.
    3. Queue a network task with *transport* to run these steps:
        1. Let *stream* be the result of creating a `WebTransportBidirectionalStream` with *internalStream* and *transport*.
        2. Enqueue *stream* to *transport*.`[[IncomingBidirectionalStreams]]`.
        3. Resolve *p* with undefined.
6. Return *p*.

To ***pullUnidirectionalStream***, given a `WebTransport` object *transport*, run these steps.

1. If *transport*.`[[State]]` is `"connecting"`, then return the result of performing the following steps upon fulfillment of *transport*.`[[Ready]]`:
    1. Return the result of pullUnidirectionalStream with *transport*.
2. If *transport*.`[[State]]` is not `"connected"`, then return a new rejected promise with an `InvalidStateError`.
3. Let *session* be *transport*.`[[Session]]`.
4. Let *p* be a new promise.
5. Run the following steps in parallel:
    1. Wait until there is an available incoming unidirectional stream.
    2. Let *internalStream* be the result of receiving an incoming unidirectional stream.
    3. Queue a network task with *transport* to run these steps:

1. Let *stream* be the result of creating a `WebTransportReceiveStream` with *internalStream* and *transport*.
2. Enqueue *stream* to *transport*.`[[IncomingUnidirectionalStreams]]`.
3. Resolve *p* with undefined.

6. Return *p*.

§ 6.3. Attributes

***`ready`*, of type Promise<undefined>, readonly**
On getting, it MUST return this's `[[Ready]]`.

***`closed`*, of type Promise<WebTransportCloseInfo>, readonly**
On getting, it MUST return this's `[[Closed]]`.

***`draining`*, of type Promise<undefined>, readonly**
On getting, it MUST return this's `[[Draining]]`.

***`datagrams`*, of type WebTransportDatagramDuplexStream, readonly**
A single duplex stream for sending and receiving datagrams over this session. The getter steps for the `datagrams` attribute SHALL be:

1. Return this's `[[Datagrams]]`.

***`incomingBidirectionalStreams`*, of type ReadableStream, readonly**
Returns a `ReadableStream` of `WebTransportBidirectionalStream`s that have been received from the server.

NOTE: Whether the incoming streams already have data on them will depend on server behavior.

The getter steps for the `incomingBidirectionalStreams` attribute SHALL be:

1. Return this's `[[IncomingBidirectionalStreams]]`.

***`incomingUnidirectionalStreams`*, of type ReadableStream, readonly**
A `ReadableStream` of unidirectional streams, each represented by a `WebTransportReceiveStream`, that have been received from the server.

NOTE: Whether the incoming streams already have data on them will depend on server behavior.

The getter steps for `incomingUnidirectionalStreams` are:

1. Return this.`[[IncomingUnidirectionalStreams]]`.

***`reliability`*, of type WebTransportReliabilityMode, readonly**
Whether connection supports unreliable (over UDP) transport or only reliable (over TCP fallback) transport. Returns `"pending"` until a connection has been established. The getter steps are to return this's `[[Reliability]]`.

***`congestionControl`*, of type WebTransportCongestionControl, readonly**
The application's preference, if requested in the constructor, and satisfied by the user agent, for a congestion control algorithm optimized for either throughput or low latency for sending on this connection. If a preference was requested but not satisfied, then the value is `"default"` The getter steps are to return this's `[[CongestionControl]]`.

***`supportsReliableOnly`*, of type boolean, readonly**
Returns true if the user agent supports WebTransport sessions over exclusively reliable connections, otherwise false.

***anticipatedConcurrentIncomingUnidirectionalStreams*, of type unsigned short, nullable**

Optionally lets an application specify the number of concurrently open incoming unidirectional streams it anticipates the server creating. If not null, the user agent SHOULD attempt to reduce future round-trips by taking `[[AnticipatedConcurrentIncomingUnidirectionalStreams]]` into consideration in its negotiations with the server.

The getter steps are to return this's `[[AnticipatedConcurrentIncomingUnidirectionalStreams]]`.

The setter steps, given *value*, are to set this's `[[AnticipatedConcurrentIncomingUnidirectionalStreams]]` to *value*.

***anticipatedConcurrentIncomingBidirectionalStreams*, of type unsigned short, nullable**

Optionally lets an application specify the number of concurrently open bidirectional streams it anticipates the server creating. If not null, the user agent SHOULD attempt to reduce future round-trips by taking `[[AnticipatedConcurrentIncomingBidirectionalStreams]]` into consideration in its negotiations with the server.

The getter steps are to return this's `[[AnticipatedConcurrentIncomingBidirectionalStreams]]`.

The setter steps, given *value*, are to set this's `[[AnticipatedConcurrentIncomingBidirectionalStreams]]` to *value*.

NOTE: Setting `anticipatedConcurrentIncomingUnidirectionalStreams` or `anticipatedConcurrentIncomingBidirectionalStreams` does not guarantee the application will receive the number of streams it anticipates.

***protocol*, of type DOMString, readonly**

Once a WebTransport session has been established and the `protocols` constructor option was used to provide a non-empty array, returns the application-level protocol selected by the server, if any. Otherwise, an empty string. The getter steps are to return this's `[[Protocol]]`.

§ 6.4. Methods

close(closeInfo)

Terminates the WebTransport session associated with the WebTransport object.

When close is called, the user agent MUST run the following steps:

1. Let *transport* be this.
2. If *transport*.`[[State]]` is `"closed"` or `"failed"`, then abort these steps.
3. If *transport*.`[[State]]` is `"connecting"`:
    1. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    2. Cleanup *transport* with *error*.
    3. Abort these steps.
4. Let *session* be *transport*.`[[Session]]`.
5. Let *code* be *closeInfo*.`closeCode`.

6. Let *reasonString* be the maximal code unit prefix of *closeInfo*.`reason` where the length of the UTF-8 encoded prefix doesn't exceed 1024.
7. Let *reason* be *reasonString*, UTF-8 encoded.
8. In parallel, terminate *session* with *code* and *reason*.

   NOTE: This also resets or sends STOP_SENDING WebTransport streams contained in *transport*. `[[SendStreams]]` and `[[ReceiveStreams]]`.

9. Cleanup *transport* with `AbortError` and *closeInfo*.

`getStats()`

Gathers stats for this `WebTransport`'s underlying connection and reports the result asynchronously.

When getStats is called, the user agent MUST run the following steps:

1. Let *transport* be this.
2. Let *p* be a new promise.
3. If *transport*.`[[State]]` is `"failed"`, reject *p* with an `InvalidStateError` and abort these steps.
4. Run the following steps in parallel:
    1. If *transport*.`[[State]]` is `"connecting"`, wait until it changes.
    2. If *transport*.`[[State]]` is `"failed"`, abort these steps after queueing a network task with *transport* to reject *p* with an `InvalidStateError`.
    3. If *transport*.`[[State]]` is `"closed"`, abort these steps after queueing a network task with *transport* to resolve *p* with the most recent stats available for the connection. The exact point at which those stats are collected is implementation-defined.
    4. Gather the stats from the underlying connection, including stats on datagrams.
    5. Queue a network task with *transport* to run the following steps:
        1. Let *stats* be a new `WebTransportConnectionStats` object representing the gathered stats.
        2. Resolve *p* with *stats*.
5. Return *p*.

`exportKeyingMaterial(BufferSource label, optional BufferSource context)`

Exports keying material from a TLS Keying Material Exporter for the TLS session uniquely associated with this `WebTransport`'s underlying connection.

When `exportKeyingMaterial` is called, the user agent MUST run the following steps:

1. Let *transport* be this.
2. Let *labelLength* be *label*.byte length.
3. If *labelLength* is more than 255, return a promise rejected with a `RangeError`.
4. Let *contextLength* be 0.
5. If *context* is given, set *contextLength* to *context*.byte length.
6. If *contextLength* is more than 255, return a promise rejected with a `RangeError`.

7. Let *p* be a new promise.
8. Run the following steps in parallel, but abort when *transport*'s `[[State]]` becomes `"closed"` or `"failed"`, and instead queue a network task with *transport* to reject *p* with an `InvalidStateError`:
    1. Let *keyingMaterial* be the result of exporting TLS keying material, as defined in [WEB-TRANSPORT-HTTP3] Section 4.7, with *labelLength*, *label*, *contextLength*, and if present, *context*.
    2. Queue a network task with *transport* to resolve *p* with *keyingMaterial*.
9. Return *p*.

`createBidirectionalStream()`

Creates a `WebTransportBidirectionalStream` object for an outgoing bidirectional stream. Note that the mere creation of a stream is not immediately visible to the peer until it is used to send data.

NOTE: There is no expectation that the server will be aware of the stream until data is sent on it.

When `createBidirectionalStream` is called, the user agent MUST run the following steps:

1. Let *transport* be this.
2. If *transport*.`[[State]]` is `"closed"` or `"failed"`, return a new rejected promise with an `InvalidStateError`.
3. Let *sendGroup* be `options`'s `sendGroup`.
4. Let *sendOrder* be `options`'s `sendOrder`.
5. Let *waitUntilAvailable* be `options`'s `waitUntilAvailable`.
6. Let *p* be a new promise.
7. Run the following steps in parallel, but abort when *transport*'s `[[State]]` becomes `"closed"` or `"failed"`, and instead queue a network task with *transport* to reject *p* with an `InvalidStateError`:
    1. Let *streamId* be a new stream ID that is valid and unique for *transport*.`[[Session]]`, as defined in [QUIC] Section 19.11. If one is not immediately available due to exhaustion, either wait for it to become available if *waitUntilAvailable* is true, or if *waitUntilAvailable* is false, abort these steps after queueing a network task with *transport* to reject *p* with a `QuotaExceededError`.
    2. Let *internalStream* be the result of creating a bidirectional stream with *transport*.`[[Session]]` and *streamId*.
    3. Queue a network task with *transport* to run the following steps:
        1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, reject *p* with an `InvalidStateError` and abort these steps.
        2. Let *stream* be the result of creating a `WebTransportBidirectionalStream` with *internalStream*, *transport*, *sendGroup*, and *sendOrder*.
        3. Resolve *p* with *stream*.
8. Return *p*.

`createUnidirectionalStream()`

Creates a `WebTransportSendStream` for an outgoing unidirectional stream. Note that the mere creation of a stream is not immediately visible to the server until it is used to send data.

NOTE: There is no expectation that the server will be aware of the stream until data is sent on it.

When `createUnidirectionalStream()` method is called, the user agent MUST run the following steps:

1. Let *transport* be this.
2. If *transport*.`[[State]]` is `"closed"` or `"failed"`, return a new rejected promise with an `InvalidStateError`.
3. Let *sendGroup* be `options`'s `sendGroup`.
4. Let *sendOrder* be `options`'s `sendOrder`.
5. Let *waitUntilAvailable* be `options`'s `waitUntilAvailable`.
6. Let *p* be a new promise.
7. Run the following steps in parallel, but abort when *transport*'s `[[State]]` becomes `"closed"` or `"failed"`, and instead queue a network task with *transport* to reject *p* with an `InvalidStateError`:
    1. Let *streamId* be a new stream ID that is valid and unique for *transport*.`[[Session]]`, as defined in [QUIC] Section 19.11. If one is not immediately available due to exhaustion, either wait for it to become available if *waitUntilAvailable* is true, or if *waitUntilAvailable* is false, abort these steps after queueing a network task with *transport* to reject *p* with a `QuotaExceededError`.
    2. Let *internalStream* be the result of creating an outgoing unidirectional stream with *transport*.`[[Session]]` and *streamId*.
    3. Queue a network task with *transport* to run the following steps:
        1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, reject *p* with an `InvalidStateError` and abort these steps.
        2. Let *stream* be the result of creating a `WebTransportSendStream` with *internalStream*, *transport*, *sendGroup*, and *sendOrder*.
        3. Resolve *p* with *stream*.
8. return *p*.

`createSendGroup()`

Creates a `WebTransportSendGroup`.

When `createSendGroup()` method is called, the user agent MUST run the following steps:

1. Let *transport* be this.
2. If *transport*.`[[State]]` is `"closed"` or `"failed"`, throw an `InvalidStateError`.
3. Return the result of creating a `WebTransportSendGroup` with *transport*.

§ 6.5. Procedures

To ***cleanup*** a `WebTransport` *transport* with *error* and optionally *closeInfo*, run these steps:

1. Let *sendStreams* be a copy of *transport*.`[[SendStreams]]`.
2. Let *receiveStreams* be a copy of *transport*.`[[ReceiveStreams]]`.

3. Let *outgoingDatagramWritables* be *transport*.`[[Datagrams]]`.`[[Writables]]`.
4. Let *incomingDatagrams* be *transport*.`[[Datagrams]]`.`[[Readable]]`.
5. Let *ready* be *transport*.`[[Ready]]`.
6. Let *closed* be *transport*.`[[Closed]]`.
7. Let *incomingBidirectionalStreams* be *transport*.`[[IncomingBidirectionalStreams]]`.
8. Let *incomingUnidirectionalStreams* be *transport*.`[[IncomingUnidirectionalStreams]]`.
9. Set *transport*.`[[SendStreams]]` to an empty set.
10. Set *transport*.`[[ReceiveStreams]]` to an empty set.
11. Set *transport*.`[[Datagrams]]`.`[[OutgoingDatagramsQueue]]` to an empty queue.
12. Set *transport*.`[[Datagrams]]`.`[[IncomingDatagramsQueue]]` to an empty queue.
13. If *closeInfo* is given, then set *transport*.`[[State]]` to `"closed"`. Otherwise, set *transport*.`[[State]]` to `"failed"`.
14. For each *stream* in *sendStreams*, run the following steps:
    1. If *stream*.`[[PendingOperation]]` is not null, reject *stream*.`[[PendingOperation]]` with *error*.
    2. Error *stream* with *error*.
15. For each *stream* in *receiveStreams*, error *stream* with *error*.

    NOTE: Script authors can inject code which runs in Promise resolution synchronously. Hence from here, do not touch *transport* as it may be mutated by scripts in an unpredictable way. This applies to logic calling this procedure, too.

16. If *closeInfo* is given, then:
    1. Resolve *closed* with *closeInfo*.
    2. Assert: *ready* is settled.
    3. Close *incomingBidirectionalStreams*.
    4. Close *incomingUnidirectionalStreams*.
    5. For each *writable* in *outgoingDatagramWritables*, close *writable*.
    6. Close *incomingDatagrams*.
17. Otherwise:
    1. Reject *closed* with *error*.
    2. Set *closed*.`[[PromiseIsHandled]]` to true.
    3. Reject *ready* with *error*.
    4. Set *ready*.`[[PromiseIsHandled]]` to true.
    5. Error *incomingBidirectionalStreams* with *error*.
    6. Error *incomingUnidirectionalStreams* with *error*.
    7. For each *writable* in *outgoingDatagramWritables*, error *writable* with *error*.
    8. Error *incomingDatagrams* with *error*.

To ***queue a network task*** with a `WebTransport` *transport* and a series of steps *steps*, run these steps:

1. Queue a global task on the network task source with *transport*'s relevant global object to run *steps*.

§ 6.6. Session termination not initiated by the client

Whenever a WebTransport session which is associated with a `WebTransport` *transport* is terminated with optionally *code* and *reasonBytes*, run these steps:

1. Queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, abort these steps.
    2. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    3. Let *closeInfo* be a new `WebTransportCloseInfo`.
    4. If *code* is given, set *closeInfo*'s `closeCode` to *code*.
    5. If *reasonBytes* is given, set *closeInfo*'s `reason` to *reasonBytes*, UTF-8 decoded.

        NOTE: No language or direction metadata is available with *reasonBytes*. First-strong heuristics can be used for direction when displaying the value.
    6. Cleanup *transport* with *error* and *closeInfo*.

Whenever a `WebTransport` *transport*'s underlying connection gets a connection error, run these steps:

1. Queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, abort these steps.
    2. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
    3. Cleanup *transport* with *error*.

§ 6.7. Context cleanup steps

This specification defines ***context cleanup steps*** as the following steps, given `WebTransport` *transport*:

1. If *transport*.`[[State]]` is `"connected"`, then:
    1. Set *transport*.`[[State]]` to `"failed"`.
    2. In parallel, terminate *transport*.`[[Session]]`.
    3. Queue a network task with *transport* to run the following steps:
        1. Let *error* be a newly created `WebTransportError` whose `source` is `"session"`.
        2. Cleanup *transport* with *error*.
2. If *transport*.`[[State]]` is `"connecting"`, set *transport*.`[[State]]` to `"failed"`.

ISSUE 1 This needs to be done in workers too. See #127 and whatwg/html#6731. ¶

§ 6.8. Garbage Collection

A WebTransport object whose [[State]] is "connecting" must not be garbage collected if [[IncomingBidirectionalStreams]], [[IncomingUnidirectionalStreams]], any WebTransportReceiveStream, or [[Datagrams]].[[Readable]] are locked, or if the ready, draining, or closed promise is being observed.

A WebTransport object whose [[State]] is "connected" must not be garbage collected if [[IncomingBidirectionalStreams]], [[IncomingUnidirectionalStreams]], any WebTransportReceiveStream, or [[Datagrams]].[[Readable]] are locked, or if the draining or closed promise is being observed.

A WebTransport object whose [[State]] is "draining" must not be garbage collected if [[IncomingBidirectionalStreams]], [[IncomingUnidirectionalStreams]], any WebTransportReceiveStream, or [[Datagrams]].[[Readable]] are locked, or if the closed promise is being observed.

A WebTransport object with an established WebTransport session that has data queued to be transmitted to the network, including datagrams in [[Datagrams]].[[OutgoingDatagramsQueue]], must not be garbage collected.

If a WebTransport object is garbage collected while the underlying connection is still open, the user agent must terminate the WebTransport session with an Application Error Code of 0 and Application Error Message of "".

§ 6.9. Configuration

```
dictionary WebTransportHash {
  DOMString algorithm;
  BufferSource value;
};

dictionary WebTransportOptions {
  boolean allowPooling = false;
  boolean requireUnreliable = false;
  sequence<WebTransportHash> serverCertificateHashes;
  WebTransportCongestionControl congestionControl = "default";
  [EnforceRange] unsigned short? anticipatedConcurrentIncomingUnidirectionalStreams = null
  [EnforceRange] unsigned short? anticipatedConcurrentIncomingBidirectionalStreams = null;
  sequence<DOMString> protocols = [];
};

enum WebTransportCongestionControl {
  "default",
  "throughput",
  "low-latency",
};
```

WebTransportOptions is a dictionary of parameters that determine how the WebTransport session is established and used.

***`allowPooling`*, of type boolean, defaulting to `false`**

When set to true, the WebTransport session can be pooled, that is, its underlying connection can be shared with other WebTransport sessions.

***`requireUnreliable`*, of type boolean, defaulting to `false`**

When set to true, the WebTransport session cannot be established over an HTTP/2 connection if an HTTP/3 connection is not possible.

***`serverCertificateHashes`*, of type sequence<WebTransportHash>**

This option is only supported for transports using dedicated connections. For transport protocols that do not support this feature, having this field non-empty SHALL result in a `NotSupportedError` exception being thrown.

If supported and non-empty, the user agent SHALL deem a server certificate trusted if and only if it can successfully verify a certificate hash against `serverCertificateHashes` and satisfies custom certificate requirements. The user agent SHALL ignore any hash that uses an unknown `algorithm`. If empty, the user agent SHALL use certificate verification procedures it would use for normal fetch operations.

This cannot be used with `allowPooling`.

***`congestionControl`*, of type WebTransportCongestionControl, defaulting to `"default"`**

Optionally specifies an application's preference for a congestion control algorithm tuned for either throughput or low-latency to be used when sending data over this connection. This is a hint to the user agent.

ISSUE 2 ¶

This configuration option is considered a feature at risk due to the lack of implementation in browsers of a congestion control algorithm, at the time of writing, that optimizes for low latency.

***`anticipatedConcurrentIncomingUnidirectionalStreams`*, of type unsigned short, nullable, defaulting to `null`**

Optionally lets an application specify the number of concurrently open incoming unidirectional streams it anticipates the server creating. The user agent MUST initially allow at least 100 incoming unidirectional streams from the server. If not null, the user agent SHOULD attempt to reduce round-trips by taking `[[AnticipatedConcurrentIncomingUnidirectionalStreams]]` into consideration in its negotiations with the server.

***`anticipatedConcurrentIncomingBidirectionalStreams`*, of type unsigned short, nullable, defaulting to `null`**

Optionally lets an application specify the number of concurrently open bidirectional streams it anticipates a server creating. The user agent MUST initially allow the server to create at least 100 bidirectional streams. If not null, the user agent SHOULD attempt to reduce round-trips by taking `[[AnticipatedConcurrentIncomingBidirectionalStreams]]` into consideration in its negotiations with the server.

***`protocols`*, of type sequence<DOMString>, defaulting to `[]`**

An optionally provided array of application-level ***protocol names***. The connection will only be established if the server reports that it has selected one of these application-level protocols.

To ***compute a certificate hash***, given a *certificate*, perform the following steps:

1. Let *cert* be *certificate*, represented as a DER encoding of Certificate message defined in [RFC5280].
2. Compute the SHA-256 hash of *cert* and return the computed value.

To ***verify a certificate hash***, given a *certificate* and an array of hashes *hashes*, perform the following steps:

```
1. Let referenceHash be the result of computing a certificate hash with certificate.
2. For every hash hash in hashes:
    1. If hash.value is not null and hash.algorithm is an ASCII case-insensitive match with "sha-256":
        1. Let hashValue be the byte sequence which hash.value represents.
        2. If hashValue is equal to referenceHash, return true.
3. Return false.
```

The ***custom certificate requirements*** are as follows: the certificate MUST be an X.509v3 certificate as defined in [RFC5280], the key used in the Subject Public Key field MUST be one of the allowed public key algorithms, the current time MUST be within the validity period of the certificate as defined in Section 4.1.2.5 of [RFC5280] and the total length of the validity period MUST NOT exceed two weeks. The user agent MAY impose additional implementation-defined requirements on the certificate.

The exact list of ***allowed public key algorithms*** used in the Subject Public Key Info field (and, as a consequence, in the TLS CertificateVerify message) is implementation-defined; however, it MUST include ECDSA with the secp256r1 (NIST P-256) named group ([RFC3279], Section 2.3.5; [RFC8422]) to provide an interoperable default. It MUST NOT contain RSA keys ([RFC3279], Section 2.3.1).

§ 6.10. `WebTransportCloseInfo` Dictionary

The ***`WebTransportCloseInfo`*** dictionary includes information relating to the error code for closing a `WebTransport`. This information is used to set the error code and reason for a CONNECTION_CLOSE frame.

```
dictionary WebTransportCloseInfo {
  unsigned long closeCode = 0;
  USVString reason = "";
};
```

The dictionary SHALL have the following attributes:

`closeCode`, of type unsigned long, defaulting to `0`
The error code communicated to the peer.

`reason`, of type USVString, defaulting to `""`
The reason for closing the `WebTransport`.

§ 6.11. `WebTransportSendOptions` Dictionary

The ***`WebTransportSendOptions`*** is a base dictionary of parameters that affect how `createUnidirectionalStream`, `createBidirectionalStream`, and the `createWritable` methods behave.

```
dictionary WebTransportSendOptions {
  WebTransportSendGroup? sendGroup = null;
  long long sendOrder = 0;
};
```

The dictionary SHALL have the following attributes:

***sendGroup*, of type WebTransportSendGroup, nullable, defaulting to `null`**
An optional `WebTransportSendGroup` to group the created stream under, or null.

***sendOrder*, of type long long, defaulting to `0`**
A send order number that, if provided, opts the created stream in to participating in ***strict ordering***. Bytes currently queued on strictly ordered streams will be sent ahead of bytes currently queued on other strictly ordered streams created with lower send order numbers.

If no send order number is provided, then the order in which the user agent sends bytes from it relative to other streams is implementation-defined. User agents are strongly encouraged however to divide bandwidth fairly between all streams that aren't starved by lower send order numbers.

NOTE: This is sender-side data prioritization which does not guarantee reception order.

§ 6.12. `WebTransportSendStreamOptions` Dictionary

The ***`WebTransportSendStreamOptions`*** is a dictionary of parameters that affect how `WebTransportSendStream`s created by `createUnidirectionalStream` and `createBidirectionalStream` behave.

```
dictionary WebTransportSendStreamOptions : WebTransportSendOptions {
  boolean waitUntilAvailable = false;
};
```

The dictionary SHALL have the following attributes:

***waitUntilAvailable*, of type boolean, defaulting to `false`**
If true, the promise returned by the `createUnidirectionalStream` or `createBidirectionalStream` call will not be settled until either the underlying connection has sufficient flow control credit to create the stream, or the connection reaches a state in which no further outgoing streams are possible. If false, the promise will be rejected if no flow control window is available at the time of the call.

§ 6.13. `WebTransportConnectionStats` Dictionary

The ***`WebTransportConnectionStats`*** dictionary includes information on WebTransport-specific stats about the WebTransport session's underlying connection.

NOTE: When pooling is used, multiple WebTransport sessions pooled on the same connection all receive the same information, i.e. the information is disclosed across pooled sessions holding the same network partition key.

NOTE: Any unavailable stats will be absent from the `WebTransportConnectionStats` dictionary.

```
dictionary WebTransportConnectionStats {
  unsigned long long bytesSent = 0;
  unsigned long long packetsSent = 0;
  unsigned long long bytesLost = 0;
  unsigned long long packetsLost = 0;
  unsigned long long bytesReceived = 0;
  unsigned long long packetsReceived = 0;
  required DOMHighResTimeStamp smoothedRtt;
  required DOMHighResTimeStamp rttVariation;
  required DOMHighResTimeStamp minRtt;
  required WebTransportDatagramStats datagrams;
  unsigned long long? estimatedSendRate = null;
  boolean atSendCapacity = false;
};
```

The dictionary SHALL have the following attributes:

***`bytesSent`*, of type unsigned long long, defaulting to `0`**
: The number of bytes sent on the underlying connection, including retransmissions. Does not include UDP or any other outer framing.

***`packetsSent`*, of type unsigned long long, defaulting to `0`**
: The number of packets sent on the underlying connection, including those that are determined to have been lost.

***`bytesLost`*, of type unsigned long long, defaulting to `0`**
: The number of bytes lost on the underlying connection (does not monotonically increase, because packets that are declared lost can subsequently be received). Does not include UDP or any other outer framing.

***`packetsLost`*, of type unsigned long long, defaulting to `0`**
: The number of packets lost on the underlying connection (does not monotonically increase, because packets that are declared lost can subsequently be received).

***`bytesReceived`*, of type unsigned long long, defaulting to `0`**
: The number of total bytes received on the underlying connection, including duplicate data for streams. Does not include UDP or any other outer framing.

***`packetsReceived`*, of type unsigned long long, defaulting to `0`**
: The number of total packets received on the underlying connection, including packets that were not processable.

***`smoothedRtt`*, of type DOMHighResTimeStamp**
: The smoothed round-trip time (RTT) currently observed on the connection, as defined in [RFC9002] Section 5.3.

***`rttVariation`*, of type DOMHighResTimeStamp**
: The mean variation in round-trip time samples currently observed on the connection, as defined in [RFC9002] Section 5.3.

***`minRtt`*, of type DOMHighResTimeStamp**
: The minimum round-trip time observed on the entire connection.

***`estimatedSendRate`*, of type unsigned long long, nullable, defaulting to `null`**
: The estimated rate at which queued data will be sent by the user agent, in bits per second. This rate applies to all streams and datagrams that share a WebTransport session and is calculated by the congestion control algorithm (potentially chosen by `congestionControl`). This estimate excludes any framing overhead and represents the rate

at which an application payload might be sent. If the user agent does not currently have an estimate, the member MUST be the `null` value. The member can be `null` even if it was not `null` in previous results.

***atSendCapacity*, of type boolean, defaulting to `false`**

A value of false indicates the `estimatedSendRate` might be application limited, meaning the application is sending significantly less data than the congestion controller allows. A congestion controller might produce a poor estimate of the available network capacity while it is application limited.

A value of true indicates the application is sending data at network capacity, and the `estimatedSendRate` reflects the network capacity available to the application.

When `atSendCapacity` is `true`, the `estimatedSendRate` reflects a ceiling. As long as the application send rate is sustained, the `estimatedSendRate` will adapt to network conditions. However, `estimatedSendRate` is allowed to be `null` while `atSendCapacity` is true.

§ 6.14. `WebTransportDatagramStats` Dictionary

The ***WebTransportDatagramStats*** dictionary includes statistics on datagram transmission over the underlying connection.

```
dictionary WebTransportDatagramStats {
  unsigned long long droppedIncoming = 0;
  unsigned long long expiredIncoming = 0;
  unsigned long long expiredOutgoing = 0;
  unsigned long long lostOutgoing = 0;
};
```

The dictionary SHALL have the following attributes:

***droppedIncoming*, of type unsigned long long, defaulting to `0`**

The number of incoming datagrams that were dropped due to the application not reading from `datagrams`' `readable` before new datagrams overflow the receive queue.

***expiredIncoming*, of type unsigned long long, defaulting to `0`**

The number of incoming datagrams that were dropped due to being older than `incomingMaxAge` before they were read from `datagrams`' `readable`.

***expiredOutgoing*, of type unsigned long long, defaulting to `0`**

The number of datagrams queued for sending that were dropped due to being older than `outgoingMaxAge` before they were able to be sent.

***lostOutgoing*, of type unsigned long long, defaulting to `0`**

The number of sent datagrams that were declared lost, as defined in [RFC9002] Section 6.1.

§ 7. Interface `WebTransportSendStream`

A `WebTransportSendStream` is a `WritableStream` providing outgoing streaming features with an outgoing unidirectional or bidirectional WebTransport stream.

It is a `WritableStream` of `Uint8Array` that can be written to, to send data to the server.

```
[Exposed=(Window,Worker), SecureContext, Transferable]
interface WebTransportSendStream : WritableStream {
  attribute WebTransportSendGroup? sendGroup;
  attribute long long sendOrder;
  Promise<WebTransportSendStreamStats> getStats();
  WebTransportWriter getWriter();
};
```

A `WebTransportSendStream` is always created by the create procedure.

The `WebTransportSendStream`’s transfer steps and transfer-receiving steps are those of `WritableStream`.

§ 7.1. Attributes

***`sendGroup`*, of type WebTransportSendGroup, nullable**

The getter steps are:

1. Return this’s `[[SendGroup]]`.

The setter steps, given *value*, are:

1. If *value* is non-null, and *value*.`[[Transport]]` is not this.`[[Transport]]`, throw an `InvalidStateError`.
2. Set this.`[[SendGroup]]` to *value*.

***`sendOrder`*, of type long long**

The getter steps are:

1. Return this’s `[[SendOrder]]`.

The setter steps, given *value*, are:

1. Set this.`[[SendOrder]]` to *value*.

§ 7.2. Methods

`getStats()`

Gathers stats specific to this `WebTransportSendStream`’s performance, and reports the result asynchronously.

When getStats is called, the user agent MUST run the following steps:

1. Let *p* be a new promise.
2. Run the following steps in parallel:
    1. Gather the stats specific to this `WebTransportSendStream`.
    2. Wait for the stats to be ready.
    3. Queue a network task with *transport* to run the following steps:

1. Let *stats* be a new `WebTransportSendStreamStats` object representing the gathered stats.
2. Resolve *p* with *stats*.

3. Return *p*.

`getWriter()`

This method must be implemented in the same manner as `getWriter` inherited from `WritableStream`, except in place of creating a `WritableStreamDefaultWriter`, it must instead create a `WebTransportWriter` with this.

§ 7.3. Internal Slots

A `WebTransportSendStream` has the following internal slots.

Internal Slot	Description (*non-normative*)
`[[InternalStream]]`	An outgoing unidirectional or bidirectional WebTransport stream.
`[[PendingOperation]]`	A promise representing a pending write or close operation, or null.
`[[Transport]]`	A `WebTransport` which owns this `WebTransportSendStream`.
`[[SendGroup]]`	An optional `WebTransportSendGroup`, or null.
`[[SendOrder]]`	An optional send order number, defaulting to 0.
`[[AtomicWriteRequests]]`	An ordered set of promises, keeping track of the subset of write requests that are atomic among those queued to be processed by the underlying sink.

§ 7.4. Procedures

To ***create*** a `WebTransportSendStream`, with an outgoing unidirectional or bidirectional WebTransport stream *internalStream*, a `WebTransport` *transport*, *sendGroup*, and a *sendOrder*, run these steps:

1. Let *stream* be a new `WebTransportSendStream`, with:

   `[[InternalStream]]`
   internalStream

   `[[PendingOperation]]`
   null

   `[[Transport]]`
   transport

   `[[SendGroup]]`
   sendGroup

   `[[SendOrder]]`
   sendOrder

`[[AtomicWriteRequests]]`
An empty ordered set of promises.

2. Let *writeAlgorithm* be an action that writes *chunk* to *stream*, given *chunk*.
3. Let *closeAlgorithm* be an action that closes *stream*.
4. Let *abortAlgorithm* be an action that aborts *stream* with *reason*, given *reason*.
5. Set up *stream* with writeAlgorithm set to *writeAlgorithm*, closeAlgorithm set to *closeAlgorithm*, abortAlgorithm set to *abortAlgorithm*.
6. Let *abortSignal* be *stream*'s [[controller]].[[abortController]].[[signal]].
7. Add the following steps to *abortSignal*.
    1. Let *pendingOperation* be *stream*.`[[PendingOperation]]`.
    2. If *pendingOperation* is null, then abort these steps.
    3. Set *stream*.`[[PendingOperation]]` to null.
    4. Let *reason* be *abortSignal*'s abort reason.
    5. Let *promise* be the result of aborting stream with *reason*.
    6. Upon fulfillment of *promise*, reject *pendingOperation* with *reason*.
8. Append *stream* to *transport*.`[[SendStreams]]`.
9. Return *stream*.

To ***write*** *chunk* to a `WebTransportSendStream` *stream*, run these steps:

1. Let *transport* be *stream*.`[[Transport]]`.
2. If *chunk* is not a `BufferSource`, return a promise rejected with a `TypeError`.
3. Let *promise* be a new promise.
4. Let *bytes* be a copy of the byte sequence which *chunk* represents.
5. Set *stream*.`[[PendingOperation]]` to *promise*.
6. Let *inFlightWriteRequest* be *stream*.inFlightWriteRequest.
7. Let *atomic* be true if stream.`[[AtomicWriteRequests]]` contains *inFlightWriteRequest*, otherwise false.
8. Run the following steps in parallel:
    1. If *atomic* is true and the current flow control window is too small for *bytes* to be sent in its entirety, then abort the remaining steps and queue a network task with *transport* to run these sub-steps:
        1. Set *stream*.`[[PendingOperation]]` to null.
        2. Abort all atomic write requests on *stream*.
    2. Otherwise, send *bytes* on *stream*.`[[InternalStream]]` and wait for the operation to complete. This sending MAY be interleaved with sending of previously queued streams and datagrams, as well as streams and datagrams yet to be queued to be sent over this transport.

       The user-agent MAY have a buffer to improve the transfer performance. Such a buffer SHOULD have a fixed upper limit, to carry the backpressure information to the user of the `WebTransportSendStream`.

This sending MUST starve until all bytes queued for sending on streams with the same `[[SendGroup]]` and a higher `[[SendOrder]]`, that are neither errored nor blocked by flow control, have been sent.

We access *stream*.`[[SendOrder]]` in parallel here. User agents SHOULD respond to live updates of these values during sending, though the details are implementation-defined.

NOTE: Ordering of retransmissions is implementation-defined, but user agents are strongly encouraged to prioritize retransmissions of data with higher `[[SendOrder]]` values.

This sending MUST NOT starve otherwise, except for flow control reasons or error.

The user agent SHOULD divide bandwidth fairly between all streams that aren't starved.

NOTE: The definition of fairness here is implementation-defined.

3. If the previous step failed due to a network error, abort the remaining steps.

   NOTE: We don't reject *promise* here because we handle network errors elsewhere, and those steps reject *stream*.`[[PendingOperation]]`.

4. Otherwise, queue a network task with *transport* to run these steps:

   1. Set *stream*.`[[PendingOperation]]` to null.
   2. If *stream*.`[[AtomicWriteRequests]]` contains *inFlightWriteRequest*, remove *inFlightWriteRequest*.
   3. Resolve *promise* with undefined.

9. Return *promise*.

NOTE: The fulfillment of the promise returned from this algorithm (or, `write(chunk)`) does **NOT** necessarily mean that the chunk is acked by the server [QUIC]. It may just mean that the chunk is appended to the buffer. To make sure that the chunk arrives at the server, the server needs to send an application-level acknowledgment message.

To ***close*** a `WebTransportSendStream` *stream*, run these steps:

1. Let *transport* be *stream*.`[[Transport]]`.
2. Let *promise* be a new promise.
3. Remove *stream* from *transport*.`[[SendStreams]]`.
4. Set *stream*.`[[PendingOperation]]` to *promise*.
5. Run the following steps in parallel:
   1. Send FIN on *stream*.`[[InternalStream]]` and wait for the operation to complete.
   2. Wait for *stream*.`[[InternalStream]]` to enter the "all data committed" state. [QUIC]
   3. Queue a network task with *transport* to run these steps:
      1. Set *stream*.`[[PendingOperation]]` to null.
      2. Resolve *promise* with undefined.
6. Return *promise*.

To ***abort*** a `WebTransportSendStream` *stream* with *reason*, run these steps:

1. Let *transport* be *stream*.`[[Transport]]`.
2. Let *promise* be a new promise.
3. Let *code* be 0.
4. Remove *stream* from *transport*.`[[SendStreams]]`.
5. If *reason* is a `WebTransportError` and *reason*.`[[StreamErrorCode]]` is not null, then set *code* to *reason*.`[[StreamErrorCode]]`.
6. If *code* < 0, then set *code* to 0.
7. If *code* > 4294967295, then set *code* to 4294967295.

   NOTE: Valid values of *code* are from 0 to 4294967295 inclusive. If the underlying connection is using HTTP/3, the code will be encoded to a number in [0x52e4a40fa8db, 0x52e5ac983162] as decribed in [WEB-TRANSPORT-HTTP3].

8. Run the following steps in parallel:
   1. Reset *stream*.`[[InternalStream]]` with *code*.
   2. Queue a network task with *transport* to resolve *promise* with undefined.
9. Return *promise*.

To ***abort all atomic write requests*** on a `WebTransportSendStream` *stream*, run these steps:

1. Let *writeRequests* be *stream*.writeRequests.
2. Let *requestsToAbort* be stream.`[[AtomicWriteRequests]]`.
3. If *writeRequests* contains a promise not in *requestsToAbort*, then error *stream* with `AbortError`, and abort these steps.
4. Empty stream.`[[AtomicWriteRequests]]`.
5. For each *promise* in *requestsToAbort*, reject *promise* with `AbortError`.
6. In parallel, for each *promise* in *requestsToAbort*, abort the sending of bytes associated with *promise*.

§ 7.5. STOP_SENDING signal coming from the server

Whenever a WebTransport stream associated with a `WebTransportSendStream` *stream* gets a STOP_SENDING signal from the server, run these steps:

1. Let *transport* be *stream*.`[[Transport]]`.
2. Let *code* be the application protocol error code attached to the STOP_SENDING frame. [QUIC]

   NOTE: Valid values of *code* are from 0 to 4294967295 inclusive. If the underlying connection is using HTTP/3, the code will be encoded to a number in [0x52e4a40fa8db, 0x52e5ac983162] as decribed in [WEB-TRANSPORT-HTTP3].

3. Queue a network task with *transport* to run these steps:

1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, abort these steps.
2. Remove *stream* from *transport*.`[[SendStreams]]`.
3. Let *error* be a newly created `WebTransportError` whose `source` is `"stream"` and `streamErrorCode` is *code*.
4. If *stream*.`[[PendingOperation]]` is not null, reject *stream*.`[[PendingOperation]]` with *error*.
5. Error *stream* with *error*.

§ 7.6. `WebTransportSendStreamStats` Dictionary

The ***`WebTransportSendStreamStats`*** dictionary includes information on stats specific to one `WebTransportSendStream`.

```
dictionary WebTransportSendStreamStats {
  unsigned long long bytesWritten = 0;
  unsigned long long bytesSent = 0;
  unsigned long long bytesAcknowledged = 0;
};
```

The dictionary SHALL have the following attributes:

`bytesWritten`**, of type unsigned long long, defaulting to `0`**

The total number of bytes the application has successfully written to this `WebTransportSendStream`. This number can only increase.

`bytesSent`**, of type unsigned long long, defaulting to `0`**

An indicator of progress on how many of the application bytes written to this `WebTransportSendStream` has been sent at least once. This number can only increase, and is always less than or equal to `bytesWritten`.

NOTE: this is progress of app data sent on a single stream only, and does not include any network overhead.

`bytesAcknowledged`**, of type unsigned long long, defaulting to `0`**

An indicator of progress on how many of the application bytes written to this `WebTransportSendStream` have been sent and acknowledged as received by the server using QUIC’s ACK mechanism. Only sequential bytes up to, but not including, the first non-acknowledged byte, are counted. This number can only increase and is always less than or equal to `bytesSent`.

NOTE: This value will match `bytesSent` when the connection is over HTTP/2.

§ 8. Interface `WebTransportSendGroup`

A `WebTransportSendGroup` is an optional organizational object that tracks transmission of data spread across many individual (typically strictly ordered) `WebTransportSendStream`s.

`WebTransportSendStream`s can, at their creation or through assignment of their `sendGroup` attribute, be ***grouped*** under at most one `WebTransportSendGroup` at any time. By default, they are ***ungrouped***.

The user agent considers `WebTransportSendGroup`s as equals when allocating bandwidth for sending `WebTransportSendStream`s. Each `WebTransportSendGroup` also establishes a separate numberspace for evaluating `sendOrder` numbers.

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransportSendGroup {
  Promise<WebTransportSendStreamStats> getStats();
};
```

A `WebTransportSendGroup` is always created by the create procedure.

§ 8.1. Methods

`getStats()`

Aggregates stats from all `WebTransportSendStream`s grouped under this sendGroup, and reports the result asynchronously.

When getStats is called, the user agent MUST run the following steps:

1. Let *p* be a new promise.
2. Let *streams* be all `WebTransportSendStream`s whose `[[SendGroup]]` is this.
3. Run the following steps in parallel:
    1. Gather stream statistics from all streams in *streams*.
    2. Queue a network task with *transport* to run the following steps:
        1. Let *stats* be a new `WebTransportSendStreamStats` object representing the aggregate numbers of the gathered stats.
        2. Resolve *p* with *stats*.
4. Return *p*.

§ 8.2. Internal Slots

A `WebTransportSendGroup` has the following internal slots.

Internal Slot	**Description (*non-normative*)**
`[[Transport]]`	The `WebTransport` object owning this `WebTransportSendGroup`.

§ 8.3. Procedures

To ***create*** a `WebTransportSendGroup`, with a `WebTransport` *transport*, run these steps:

1. Let *sendGroup* be a new `WebTransportSendGroup`, with:

[[Transport]]
transport

2. Return *sendGroup*.

§ 9. Interface `WebTransportReceiveStream`

A `WebTransportReceiveStream` is a `ReadableStream` providing incoming streaming features with an incoming unidirectional or bidirectional WebTransport stream.

It is a `ReadableStream` of `Uint8Array` that can be read from, to consume data received from the server. `WebTransportReceiveStream` is a readable byte stream, and hence it allows its consumers to use a BYOB reader as well as a default reader.

```
[Exposed=(Window,Worker), SecureContext, Transferable]
interface WebTransportReceiveStream : ReadableStream {
  Promise<WebTransportReceiveStreamStats> getStats();
};
```

A `WebTransportReceiveStream` is always created by the create procedure.

The `WebTransportReceiveStream`’s transfer steps and transfer-receiving steps are those of `ReadableStream`.

§ 9.1. Methods

getStats()

Gathers stats specific to this `WebTransportReceiveStream`’s performance, and reports the result asynchronously.

When getStats is called, the user agent MUST run the following steps:

1. Let *p* be a new promise.
2. Run the following steps in parallel:
    1. Gather the stats specific to this `WebTransportReceiveStream`.
    2. Queue a network task with *transport* to run the following steps:
        1. Let *stats* be a new `WebTransportReceiveStreamStats` object representing the gathered stats.
        2. Resolve *p* with *stats*.
3. Return *p*.

§ 9.2. Internal Slots

A `WebTransportReceiveStream` has the following internal slots.

Internal Slot	Description (*non-normative*)
`[[InternalStream]]`	An incoming unidirectional or bidirectional WebTransport stream.
`[[Transport]]`	The `WebTransport` object owning this `WebTransportReceiveStream`.

§ 9.3. Procedures

To ***create*** a `WebTransportReceiveStream`, with an incoming unidirectional or bidirectional WebTransport stream *internalStream* and a `WebTransport` *transport*, run these steps:

1. Let *stream* be a new `WebTransportReceiveStream`, with:

   `[[InternalStream]]`
   internalStream

   `[[Transport]]`
   transport

2. Let *pullAlgorithm* be an action that pulls bytes from *stream*.
3. Let *cancelAlgorithm* be an action that cancels *stream* with *reason*, given *reason*.
4. Set up with byte reading support *stream* with pullAlgorithm set to *pullAlgorithm* and cancelAlgorithm set to *cancelAlgorithm*.
5. Append *stream* to *transport*.`[[ReceiveStreams]]`.
6. Return *stream*.

To ***pull bytes*** from a `WebTransportReceiveStream` *stream*, run these steps.

1. Let *transport* be *stream*.`[[Transport]]`.
2. Let *internalStream* be *stream*.`[[InternalStream]]`.
3. Let *promise* be a new promise.
4. Let *buffer*, *offset*, and *maxBytes* be null.
5. If *stream*'s current BYOB request view for *stream* is not null:
   1. Set *offset* to *stream*'s current BYOB request view.[[ByteOffset]].
   2. Set *maxBytes* to *stream*'s current BYOB request view's byte length.
   3. Set *buffer* to *stream*'s current BYOB request view's underlying buffer.
6. Otherwise:
   1. Set *offset* to 0.
   2. Set *maxBytes* to an implementation-defined size.
   3. Set *buffer* be a new `ArrayBuffer` with *maxBytes* size. If allocating the `ArrayBuffer` fails, return a promise rejected with a `RangeError`.
7. Run the following steps in parallel:

1. Write the bytes that area read from *internalStream* into *buffer* with offset *offset*, up to *maxBytes* bytes. Wait until either at least one byte is read or FIN is received. Let *read* be the number of read bytes, and let *hasReceivedFIN* be whether FIN was accompanied.

   The user-agent MAY have a buffer to improve the transfer performance. Such a buffer SHOULD have a fixed upper limit, to carry the backpressure information to the server.

   NOTE: This operation may return before filling up all of *buffer*.

2. If the previous step failed, abort the remaining steps.

   NOTE: We don't reject *promise* here because we handle network errors elsewhere, and those steps error *stream*, which rejects any read requests awaiting this pull.

3. Queue a network task with *transport* to run these steps:

   NOTE: If the buffer described above is available in the event loop where this procedure is running, the following steps may run immediately.

   1. If *read* > 0:
      1. Set *view* to a new `Uint8Array` with *buffer*, *offset* and *read*.
      2. Enqueue *view* into *stream*.
   2. If *hasReceivedFIN* is true:
      1. Remove *stream* from *transport*.`[[ReceiveStreams]]`.
      2. Close *stream*.
   3. Resolve *promise* with undefined.

8. Return *promise*.

To ***cancel*** a `WebTransportReceiveStream` *stream* with *reason*, run these steps.

1. Let *transport* be *stream*.`[[Transport]]`.
2. Let *internalStream* be *stream*.`[[InternalStream]]`.
3. Let *promise* be a new promise.
4. Let *code* be 0.
5. If *reason* is a `WebTransportError` and *reason*.`[[StreamErrorCode]]` is not null, then set *code* to *reason*.`[[StreamErrorCode]]`.
6. If *code* < 0, then set *code* to 0.
7. If *code* > 4294967295, then set *code* to 4294967295.

   NOTE: Valid values of *code* are from 0 to 4294967295 inclusive. If the underlying connection is using HTTP/3, the code will be encoded to a number in [0x52e4a40fa8db, 0x52e5ac983162] as decribed in [WEB-TRANSPORT-HTTP3].

8. Remove *stream* from *transport*.`[[SendStreams]]`.

9. Run the following steps in parallel:
    1. Send STOP_SENDING with *internalStream* and *code*.
    2. Queue a network task with *transport* to run these steps:

        NOTE: If the buffer described above is available in the event loop where this procedure is running, the following steps may run immediately.

        1. Remove *stream* from *transport*.`[[ReceiveStreams]]`.
        2. Resolve *promise* with undefined.
10. Return *promise*.

§ 9.4. Reset signal coming from the server

Whenever a WebTransport stream associated with a `WebTransportReceiveStream` *stream* gets a RESET_STREAM signal from the server, run these steps:

1. Let *transport* be *stream*.`[[Transport]]`.
2. Let *code* be the application protocol error code attached to the RESET_STREAM frame. [QUIC]

    NOTE: Valid values of *code* are from 0 to 4294967295 inclusive. If the underlying connection is using HTTP/3, the code will be encoded to a number in [0x52e4a40fa8db, 0x52e5ac983162] as decribed in [WEB-TRANSPORT-HTTP3].

3. Queue a network task with *transport* to run these steps:
    1. If *transport*.`[[State]]` is `"closed"` or `"failed"`, abort these steps.
    2. Remove *stream* from *transport*.`[[ReceiveStreams]]`.
    3. Let *error* be a newly created `WebTransportError` whose `source` is `"stream"` and `streamErrorCode` is *code*.
    4. Error *stream* with *error*.

§ 9.5. `WebTransportReceiveStreamStats` Dictionary

The ***`WebTransportReceiveStreamStats`*** dictionary includes information on stats specific to one `WebTransportReceiveStream`.

```
dictionary WebTransportReceiveStreamStats {
  unsigned long long bytesReceived = 0;
  unsigned long long bytesRead = 0;
};
```

The dictionary SHALL have the following attributes:

bytesReceived, **of type unsigned long long, defaulting to `0`**

An indicator of progress on how many of the server application's bytes intended for this `WebTransportReceiveStream` have been received so far. Only sequential bytes up to, but not including, the first missing byte, are counted. This number can only increase.

NOTE: this is progress of app data received on a single stream only, and does not include any network overhead.

bytesRead, **of type unsigned long long, defaulting to `0`**

The total number of bytes the application has successfully read from this `WebTransportReceiveStream`. This number can only increase, and is always less than or equal to `bytesReceived`.

§ 10. Interface `WebTransportBidirectionalStream`

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransportBidirectionalStream {
  readonly attribute WebTransportReceiveStream readable;
  readonly attribute WebTransportSendStream writable;
};
```

§ 10.1. Internal slots

A `WebTransportBidirectionalStream` has the following internal slots.

Internal Slot	Description (*non-normative*)
`[[Readable]]`	A `WebTransportReceiveStream`.
`[[Writable]]`	A `WebTransportSendStream`.
`[[Transport]]`	The `WebTransport` object owning this `WebTransportBidirectionalStream`.

§ 10.2. Attributes

readable, **of type WebTransportReceiveStream, readonly**

The getter steps are to return this's `[[Readable]]`.

writable, **of type WebTransportSendStream, readonly**

The getter steps are to return this's `[[Writable]]`.

§ 10.3. Procedures

To ***create*** a `WebTransportBidirectionalStream` with a bidirectional WebTransport stream *internalStream*, a `WebTransport` object *transport*, and a *sendOrder*, run these steps.

1. Let *readable* be the result of creating a `WebTransportReceiveStream` with *internalStream* and *transport*.

2. Let *writable* be the result of creating a `WebTransportSendStream` with *internalStream*, *transport*, and *sendOrder*.
3. Let *stream* be a new `WebTransportBidirectionalStream`, with:

   `[[Readable]]`
   : *readable*

   `[[Writable]]`
   : *writable*

   `[[Transport]]`
   : *transport*

4. Return *stream*.

§ 11. `WebTransportWriter` Interface

`WebTransportWriter` is a subclass of `WritableStreamDefaultWriter` that adds one method.

A `WebTransportWriter` is always created by the create procedure.

```
[Exposed=*, SecureContext]
interface WebTransportWriter : WritableStreamDefaultWriter {
  Promise<undefined> atomicWrite(optional any chunk);
};
```

§ 11.1. Methods

`atomicWrite(chunk)`

The `atomicWrite` method will reject if the *chunk* given to it could not be sent in its entirety within the flow control window that is current at the time of sending. This behavior is designed to satisfy niche transactional applications sensitive to flow control deadlocks ([RFC9308] Section 4.4).

NOTE: `atomicWrite` can still reject after sending some data. Though it provides atomicity with respect to flow control, other errors may occur. `atomicWrite` does not prevent data from being split between packets or being interleaved with other data. Only the sender learns if `atomicWrite` fails due to lack of available flow control credit.

NOTE: Atomic writes can still block if queued behind non-atomic writes. If the atomic write is rejected, everything queued behind it at that moment will be rejected as well. Any non-atomic writes rejected in this way will error the stream. Applications are therefore encouraged to always await atomic writes.

When `atomicWrite` is called, the user agent MUST run the following steps:

1. Let *p* be the result of `write(chunk)` on `WritableStreamDefaultWriter` with *chunk*.
2. Append *p* to *stream*.`[[AtomicWriteRequests]]`.
3. Return the result of reacting to *p* with the following steps:
   1. If *stream*.`[[AtomicWriteRequests]]` contains *p*, remove *p*.

2. If *p* was rejected with reason *r*, then return a promise rejected with *r*.
3. Return undefined.

§ 11.2. Procedures

To ***create*** a `WebTransportWriter`, with a `WebTransportSendStream` *stream*, run these steps:

1. Let *writer* be a new `WebTransportWriter`.
2. Run the new WritableStreamDefaultWriter(stream) constructor steps passing *writer* as this, and *stream* as the constructor argument.
3. Return *writer*.

§ 12. `WebTransportError` Interface

`WebTransportError` is a subclass of `DOMException` that represents

- An error coming from the server or the network, or
- A reason for a client-initiated abort operation.

```
[Exposed=(Window,Worker), Serializable, SecureContext]
interface WebTransportError : DOMException {
  constructor(optional DOMString message = "", optional WebTransportErrorOptions options =

  readonly attribute WebTransportErrorSource source;
  readonly attribute unsigned long? streamErrorCode;
};

dictionary WebTransportErrorOptions {
  WebTransportErrorSource source = "stream";
  [Clamp] unsigned long? streamErrorCode = null;
};

enum WebTransportErrorSource {
  "stream",
  "session",
};
```

§ 12.1. Internal slots

A `WebTransportError` has the following internal slots.

Internal Slot	Description (*non-normative*)
`[[Source]]`	A `WebTransportErrorSource` indicating the source of this error.

Internal Slot	Description (*non-normative*)
[[StreamErrorCode]]	The application protocol error code for this error, or null.

§ 12.2. Constructor

The ***new WebTransportError(message, options)*** constructor steps are:

1. Set *this*'s name to `"WebTransportError"`.
2. Set *this*'s message to *message*.
3. Set *this*'s internal slots as follows:

   `[[Source]]`
   : *options*.`source`

   `[[StreamErrorCode]]`
   : *options*.`streamErrorCode`

   NOTE: This name does not have a mapping to a legacy code, so this's `code` is 0.

§ 12.3. Attributes

source, **of type WebTransportErrorSource, readonly**
: The getter steps are to return this's `[[Source]]`.

streamErrorCode, **of type unsigned long, readonly, nullable**
: The getter steps are to return this's `[[StreamErrorCode]]`.

§ 12.4. Serialization

`WebTransportError` objects are serializable objects. Their serialization steps, given *value* and *serialized*, are:

1. Run the `DOMException` serialization steps given *value* and *serialized*.
2. Set *serialized*.`[[Source]]` to *value*.`[[Source]]`.
3. Set *serialized*.`[[StreamErrorCode]]` to *value*.`[[StreamErrorCode]]`.

Their deserialization steps, given *serialized* and *value*, are:

1. Run the `DOMException` deserialization steps given *serialized* and *value*.
2. Set *value*.`[[Source]]` to *serialized*.`[[Source]]`.
3. Set *value*.`[[StreamErrorCode]]` *serialized*.`[[StreamErrorCode]]`.

§ 13. Protocol Mappings

This section is non-normative.

This section describes the underlying protocol behavior of methods defined in this specification, utilizing [WEB-TRANSPORT-OVERVIEW]. Cause and effect may not be immediate due to buffering.

WebTransport Protocol Action	API Effect
received WT_DRAIN_SESSION	await wt.draining

If the underlying connection is using HTTP/3, the following protocol behaviors from [WEB-TRANSPORT-HTTP3] apply.

The application streamErrorCode in the WebTransportError error is converted to an httpErrorCode, and vice versa, as specified in [WEB-TRANSPORT-HTTP3] Section 4.3.

API Method	QUIC Protocol Action
writable.abort(error)	sends RESET_STREAM with httpErrorCode
writable.close()	sends STREAM with FIN bit set
writable.getWriter().write(chunk)()	sends STREAM
writable.getWriter().close()	sends STREAM with FIN bit set
writable.getWriter().abort(error)	sends RESET_STREAM with httpErrorCode
readable.cancel(error)	sends STOP_SENDING with httpErrorCode
readable.getReader().cancel(error)	sends STOP_SENDING with httpErrorCode
wt.close(closeInfo)	terminates session with closeInfo

QUIC Protocol Action	API Effect
received STOP_SENDING with httpErrorCode	errors writable with streamErrorCode
received STREAM	(await readable.getReader().read()).value
received STREAM with FIN bit set	(await readable.getReader().read()).done
received RESET_STREAM with httpErrorCode	errors readable with streamErrorCode
Session cleanly terminated with closeInfo	(await wt.closed).closeInfo, and errors open streams
Network error	(await wt.closed) rejects, and errors open streams

NOTE: As discussed in [QUIC] Section 3.2, receipt of a RESET_STREAM frame is not always indicated to the application. Receipt of the RESET_STREAM can be signaled immediately, interrupting delivery of stream data with any data not consumed being discarded. However, immediate signaling is not required. Also, if stream data is completely received but has not yet been read by the application, the RESET_STREAM signal can be suppressed.

HTTP/3 Protocol Action	API Effect
received GOAWAY	await wt.draining

If the underlying connection is using HTTP/2, the following protocol behaviors from [WEB-TRANSPORT-HTTP2] apply. Note that, unlike for HTTP/3, the stream error code does not need to be converted to an HTTP error code, and vice versa.

API Method	HTTP/2 Protocol Action
writable.abort(error)	sends WT_RESET_STREAM with error
writable.close()	sends WT_STREAM with FIN bit set
writable.getWriter().write()	sends WT_STREAM
writable.getWriter().close()	sends WT_STREAM with FIN bit set
writable.getWriter().abort(error)	sends WT_RESET_STREAM with error
readable.cancel(error)	sends WT_STOP_SENDING with error
readable.getReader().cancel(error)	sends WT_STOP_SENDING with error
wt.close(closeInfo)	terminates session with closeInfo

HTTP/2 Protocol Action	API Effect
received WT_STOP_SENDING with error	errors writable with streamErrorCode
received WT_STREAM	(await readable.getReader().read()).value
received WT_STREAM with FIN bit set	(await readable.getReader().read()).done
received WT_RESET_STREAM with error	errors readable with streamErrorCode
Session cleanly terminated with closeInfo	(await wt.closed).closeInfo, and errors open streams
Network error	(await wt.closed) rejects, and errors open streams
received GOAWAY	await wt.draining

§ 14. Privacy and Security Considerations

This section is non-normative; it specifies no new behaviour, but instead summarizes information already present in other parts of the specification.

§ 14.1. Confidentiality of Communications

The fact that communication is taking place cannot be hidden from adversaries that can observe the network, so this has to be regarded as public information.

All of the transport protocols described in this document use either TLS [RFC8446] or a semantically equivalent protocol, thus providing all of the security properties of TLS, including confidentiality and integrity of the traffic. WebTransport over HTTP uses the same certificate verification mechanism as outbound HTTP requests, thus relying on the same public key infrastructure for authentication of the remote server. In WebTransport, certificate verification errors are fatal; no interstitial allowing bypassing certificate validation is available.

§ 14.2. State Persistence

WebTransport does not by itself create any new unique identifiers or new ways to persistently store state, nor does it automatically expose any of the existing persistent state to the server. For instance, neither [WEB-TRANSPORT-HTTP3] nor [WEB-TRANSPORT-HTTP2] send cookies or support HTTP authentication or caching invalidation mechanisms. Since they do use TLS, they inherit TLS persistent state such as TLS session tickets, which while not visible to passive network observers, could be used by the server to correlate different connections from the same client.

§ 14.3. Protocol Security

WebTransport imposes a set of requirements as described in [WEB-TRANSPORT-OVERVIEW], including:

1. Ensuring that the remote server is aware that the WebTransport protocol is in use and confirming that the remote server is willing to use the WebTransport protocol. [WEB-TRANSPORT-HTTP3] uses a combination of ALPN [RFC7301], an HTTP/3 setting, and a `:protocol` pseudo-header to identify the WebTransport protocol. [WEB-TRANSPORT-HTTP2] uses a combination of ALPN, an HTTP/2 setting, and a `:protocol` pseudo-header to identify the WebTransport protocol.
2. Allowing the server to filter connections based on the origin of the resource originating the transport session. The `Origin` header field on the session establishment request carries this information.

Protocol security related considerations are described in the *Security Considerations* sections of [WEB-TRANSPORT-HTTP3] and [WEB-TRANSPORT-HTTP2].

Networking APIs can be commonly used to scan the local network for available hosts, and thus be used for fingerprinting and other forms of attacks. WebTransport follows the WebSocket approach to this problem: the specific connection error is not returned until an endpoint is verified to be a WebTransport endpoint; thus, the Web application cannot distinguish between a non-existing endpoint and the endpoint that is not willing to accept connections from the Web.

§ 14.4. Authentication using Certificate Hashes

Normally, a user agent authenticates a TLS connection between itself and a remote endpoint by verifying the validity of the TLS server certificate provided against the server name in the URL [RFC9525]. This is accomplished by chaining server certificates to one of the trust anchors maintained by the user agent; the trust anchors in question are responsible for authenticating the server names in the certificates. We will refer to this system as Web PKI.

This API provides web applications with a capability to connect to a remote network endpoint authenticated by a specific server certificate, rather than its server name. This mechanism enables connections to endpoints for which getting long-term certificates can be challenging, including hosts that are ephemeral in nature (e.g. short-lived virtual machines), or that are not publicly routable. Since this mechanism substitutes Web PKI-based authentication for an individual connection, we need to compare the security properties of both.

A remote server will be able to successfully perform a TLS handshake only if it posesses the private key corresponding to the public key of the certificate specified. The API identifies the certificates using their hashes. That is only secure as long as the cryptographic hash function used has second-preimage resistance. The only function defined in this document is SHA-256; the API provides a way to introduce new hash functions through allowing multiple algorithm-hash pairs to be specified.

It is important to note that Web PKI provides additional security mechanisms in addition to simply establishing a chain of trust for a server name. One of them is handling certificate revocation. In cases where the certificate used is ephemeral, such a mechanism is not necessary. In other cases, the Web application has to consider the mechanism by which the certificate hashes are provisioned; for instance, if the hash is provided as a cached HTTP resource, the cache needs to be invalidated if the corresponding certificate has been rotated due to compromise. Another security feature provided by the Web PKI are safeguards against certain issues with key generation, such as rejecting certificates with known weak keys; while this specification does not provide any specific guidance, browsers MAY reject those as a part of implementation-defined behavior.

Web PKI enforces an expiry period requirement on the certificates. This requirement limits the scope of potential key compromise; it also forces server operators to design systems that support and actively perform key rotation. For this reason, WebTransport imposes a similar expiry requirement; as the certificates are expected to be ephemeral or short-lived, the expiry period is limited to two weeks. The two weeks limit is a balance between setting the expiry limit as low as possible to minimize consequences of a key compromise, and maintaining it sufficiently high to accomodate for clock skew across devices, and to lower the costs of synchronizing certificates between the client and the server side.

The WebTransport API lets the application specify multiple certificate hashes at once, allowing the client to accept multiple certificates for a period in which a new certificate is being rolled out.

Unlike a similar mechanism in WebRTC, the server certificate hash API in WebTransport does not provide any means of authenticating the client; the fact that the client knows what the server certificate is or how to contact it is not sufficient. The application has to establish the identity of the client in-band if necessary.

§ 14.5. Fingerprinting and Tracking

This API provides sites with the ability to generate network activity and closely observe the effect of this activity. The information obtained in this way might be identifying.

It is important to recognize that very similar networking capabilities are provided by other web platform APIs (such as fetch and [webrtc]). The net adverse effect on privacy due to adding WebTransport is therefore minimal. The considerations in this section applies equally to other networking capabilities.

Measuring network characteristics requires that the network be used and that the effect of that usage be measured, both of which are enabled by this API. WebTransport provides sites with an ability to generate network activity toward a server of their choice and observe the effects. Observations of both the stable properties of a network path and dynamic effect of network usage are possible.

Information about the network is available to the server either directly through its own networking stack, indirectly through the rate at which data is consumed or transmitted by the client, or as part of the statistics provided by the API (see § 6.13 WebTransportConnectionStats Dictionary). Consequently, restrictions on information in user agents is not the only mechanism that might be needed to manage these privacy risks.

§ 14.5.1. Static Observations

A site can observe available network capacity or round trip time (RTT) between a user agent and a chosen server. This information can be identifying when combined with other tracking vectors. RTT can also reveal something about the physical location of a user agent, especially if multiple measurements can be made from multiple vantage points.

Though networking is shared, network use is often sporadic, which means that sites are often able to observe the capacity and round trip times of an uncontested or lightly loaded network path. These properties are stable for many people as their network location does not change and the position of network bottlenecks--which determine available capacity--can be close to a user agent.

§ 14.5.2. Shared Networking

Contested links present sites with opportunities to enable cross-site recognition, which might be used to perform unsanctioned tracking [UNSANCTIONED-TRACKING]. Network capacity is a finite shared resource, so a user agent that concurrently accesses different sites might reveal a connection between the identity presented to each site.

The use of networking capabilities on one site reduces the capacity available to other sites, which can be observed using networking APIs. Network usage and metrics can change dynamically, so any change can be observed in real time. This might allow sites to increase confidence that activity on different sites originates from the same user.

A user agent could limit or degrade access to feedback mechanisms such as statistics (§ 6.13 WebTransportConnectionStats Dictionary) for sites that are inactive or do not have focus (*HTML* § 6.6 Focus). As noted, this does not prevent a server from making observations about changes in the network.

§ 14.5.3. Pooled Sessions

Similar to shared networking scenarios, when sessions are pooled on a single connection, information from one session is affected by the activity of another session. One session could infer information about the activity of another session, such as the rate at which another application is sending data.

The use of a shared connection already allows the server to correlate sessions. Use of a network partition key disables pooling where use of a shared session might enable unwanted cross-site recognition.

§ 15. Examples

§ 15.1. Sending a buffer of datagrams

This section is non-normative.

Sending a buffer of datagrams can be achieved by using the `datagrams`' `createWritable` method and the resulting stream's writer. In the following example datagrams are only sent if the transport is ready to send.

```
EXAMPLE 1
async function sendDatagrams(url, datagrams) {
  const wt = new WebTransport(url);
  const writable = wt.datagrams.createWritable();
  const writer = writable.getWriter();
  for (const bytes of datagrams) {
    await writer.ready;
    writer.write(bytes).catch(() => {});
  }
  await writer.close();
}
```

§ 15.2. Sending datagrams at a fixed rate

This section is non-normative.

Sending datagrams at a fixed rate regardless if the transport is ready to send can be achieved by simply using `datagrams`' `createWritable` method and the resulting stream's writer without awaiting the `ready` attribute.

```
EXAMPLE 2
// Sends datagrams every 100 ms.
async function sendFixedRate(url, createDatagram, ms = 100) {
  const wt = new WebTransport(url);
  const writable = wt.datagrams.createWritable();
  const writer = writable.getWriter();
  const bytes = createDatagram();
  setInterval(() => writer.write(bytes).catch(() => {}), ms);
}
```

§ 15.3. Receiving datagrams

This section is non-normative.

Datagrams can be received by reading from the transport.`datagrams`.`readable` attribute. Null values may indicate that packets are not being processed quickly enough.

```
EXAMPLE 3
async function receiveDatagrams(url) {
  const wt = new WebTransport(url);
  for await (const datagram of wt.datagrams.readable) {
    // Process the datagram
  }
}
```

§ 15.4. Receiving datagrams with a BYOB reader

This section is non-normative.

As `datagrams` are readable byte streams, you can acquire a BYOB reader for them, which allows more precise control over buffer allocation in order to avoid copies. This example reads the datagram into a 64kB memory buffer.

EXAMPLE 4

```
const wt = new WebTransport(url);

for await (const datagram of wt.datagrams.readable) {
  const reader = datagram.getReader({ mode: "byob" });

  let array_buffer = new ArrayBuffer(65536);
  const buffer = await readInto(array_buffer);
}

async function readInto(buffer) {
  let offset = 0;

  while (offset < buffer.byteLength) {
    const {value: view, done} = await reader.read(
        new Uint8Array(buffer, offset, buffer.byteLength - offset));
    buffer = view.buffer;
    if (done) {
      break;
    }
    offset += view.byteLength;
  }

  return buffer;
}
```

§ 15.5. Sending a stream

This section is non-normative.

Sending data as a one-way stream can be achieved by using the `createUnidirectionalStream` function and the resulting stream’s writer.

The written chunk boundaries aren’t preserved on reception, as the bytes might coalesce on the wire. Applications are therefore encouraged to provide their own framing.

EXAMPLE 5

```
async function sendData(url, ...data) {
  const wt = new WebTransport(url);
  const writable = await wt.createUnidirectionalStream();
  const writer = writable.getWriter();
  for (const bytes of data) {
    await writer.ready;
    writer.write(bytes).catch(() => {});
  }
  await writer.close();
}
```

The streams spec discourages awaiting the promise from write().

Encoding can also be done through pipes from a `ReadableStream`, for example using `TextEncoderStream`.

EXAMPLE 6

```
async function sendText(url, readableStreamOfTextData) {
  const wt = new WebTransport(url);
  const writable = await wt.createUnidirectionalStream();
  await readableStreamOfTextData
    .pipeThrough(new TextEncoderStream("utf-8"))
    .pipeTo(writable);
}
```

§ 15.6. Receiving incoming streams

This section is non-normative.

Reading incoming streams can be achieved by iterating over the `incomingUnidirectionalStreams` attribute, and then consuming each `WebTransportReceiveStream` by iterating over its chunks.

Chunking is determined by the user agent, not the sender.

EXAMPLE 7 ¶
```
async function receiveData(url, processTheData) {
  const wt = new WebTransport(url);
  for await (const readable of wt.incomingUnidirectionalStreams) {
    // consume streams individually using IFFEs, reporting per-stream errors
    ((async () => {
      try {
        for await (const bytes of readable) {
          processTheData(bytes);
        }
      } catch (e) {
        console.error(e);
      }
    })());
  }
}
```

Decoding can also be done through pipes to new WritableStreams, for example using `TextDecoderStream`. This example assumes text output should not be interleaved, and therefore only reads one stream at a time.

EXAMPLE 8 ¶
```
async function receiveText(url, createWritableStreamForTextData) {
  const wt = new WebTransport(url);
  for await (const readable of wt.incomingUnidirectionalStreams) {
    // consume sequentially to not interleave output, reporting per-stream errors
    try {
      await readable
       .pipeThrough(new TextDecoderStream("utf-8"))
       .pipeTo(createWritableStreamForTextData());
    } catch (e) {
      console.error(e);
    }
  }
}
```

§ 15.7. Receiving a stream with a BYOB reader

This section is non-normative.

As `WebTransportReceiveStream`s are readable byte streams, you can acquire a BYOB reader for them, which allows more precise control over buffer allocation in order to avoid copies. This example reads the first 1024 bytes from a `WebTransportReceiveStream` into a single memory buffer.

EXAMPLE 9

```
const wt = new WebTransport(url);

const reader = wt.incomingUnidirectionalStreams.getReader();
const { value: recv_stream, done } = await reader.read();
const byob_reader = recv_stream.getReader({ mode: "byob" });

let array_buffer = new ArrayBuffer(1024);
const buffer = await readInto(array_buffer);

async function readInto(buffer) {
  let offset = 0;

  while (offset < buffer.byteLength) {
    const {value: view, done} = await reader.read(
        new Uint8Array(buffer, offset, buffer.byteLength - offset));
    buffer = view.buffer;
    if (done) {
      break;
    }
    offset += view.byteLength;
  }

  return buffer;
}
```

§ 15.8. Sending a transactional chunk on a stream

This section is non-normative.

Sending a transactional piece of data on a unidirectional stream, only if it can be done entirely without blocking on flow control, can be achieved by using the `getWriter` function and the resulting writer.

```
EXAMPLE 10
async function sendTransactionalData(wt, bytes) {
  const writable = await wt.createUnidirectionalStream();
  const writer = writable.getWriter();
  await writer.ready;
  try {
    await writer.atomicWrite(bytes);
  } catch (e) {
    if (e.name != "AbortError") throw e;
    // rejected to avoid blocking on flow control
    // The writable remains un-errored provided no non-atomic writes are pending
  } finally {
    writer.releaseLock();
  }
}
```

§ 15.9. Using a server certificate hash

This section is non-normative.

A WebTransport session can override the default trust evaluation performed by the client with a check against the hash of the certificate provided to the server. In the example below, `hashValue` is a `BufferSource` containing the SHA-256 hash of a server certificate that the underlying connection should consider to be valid.

```
EXAMPLE 11
const wt = new WebTransport(url, {
  serverCertificateHashes: [
    {
      algorithm: "sha-256",
      value: hashValue,
    }
  ]
});
await wt.ready;
```

§ 15.10. Complete example

This section is non-normative.

This example illustrates use of the closed and ready promises, opening of uni-directional and bi-directional streams by either the client or the server, and sending and receiving datagrams.

The `writable` attribute that used to exist on a transport's `datagrams` is easy to polyfill as follows:

```
wt.datagrams.writable ||= wt.datagrams.createWritable();
```

EXAMPLE 12

```
// Adds an entry to the event log on the page, optionally applying a specified
// CSS class.

let wt, streamNumber, datagramWriter;

connect.onclick = async () => {
  try {
    const url = document.getElementById('url').value;

    wt = new WebTransport(url);
    wt.datagrams.writable ||= wt.datagrams.createWritable();
    addToEventLog('Initiating connection...');
    await wt.ready;
    addToEventLog(`${(wt.reliability == "reliable-only")? "TCP" : "UDP"} ` +
                  `connection ready.`);
    wt.closed
      .then(() => addToEventLog('Connection closed normally.'))
      .catch(() => addToEventLog('Connection closed abruptly.', 'error'));

    streamNumber = 1;
    datagramWriter = wt.datagrams.writable.getWriter();

    readDatagrams();
    acceptUnidirectionalStreams();
    document.forms.sending.elements.send.disabled = false;
    document.getElementById('connect').disabled = true;
  } catch (e) {
    addToEventLog(`Connection failed. ${e}`, 'error');
  }
}

sendData.onclick = async () => {
  const form = document.forms.sending.elements;
  const data = sending.data.value;
  const bytes = new TextEncoder('utf-8').encode(data);
  try {
    switch (form.sendtype.value) {
      case 'datagram': {
        await datagramWriter.ready;
        datagramWriter.write(bytes).catch(() => {});
        addToEventLog(`Sent datagram: ${data}`);
        break;
      }
      case 'unidi': {
        const writable = await wt.createUnidirectionalStream();
        const writer = writable.getWriter();
        writer.write(bytes).catch(() => {});
        await writer.close();
        addToEventLog(`Sent a unidirectional stream with data: ${data}`);
        break;
      }
      case 'bidi': {
        const duplexStream = await wt.createBidirectionalStream();
        const n = streamNumber++;
        readFromIncomingStream(duplexStream.readable, n);

        const writer = duplexStream.writable.getWriter();
        writer.write(bytes).catch(() => {});
        await writer.close();
        addToEventLog(`Sent bidirectional stream #${n} with data: ${data}`);
        break;
      }
    }
  } catch (e) {
    addToEventLog(`Error while sending data: ${e}`, 'error');
  }
}

// Reads datagrams into the event log until EOF is reached.
async function readDatagrams() {
  try {
    const decoder = new TextDecoderStream('utf-8');

    for await (const data of wt.datagrams.readable.pipeThrough(decoder)) {
      addToEventLog(`Datagram received: ${data}`);
    }
    addToEventLog('Done reading datagrams!');
  } catch (e) {
    addToEventLog(`Error while reading datagrams: ${e}`, 'error');
  }
}

async function acceptUnidirectionalStreams() {
  try {
    for await (const readable of wt.incomingUnidirectionalStreams) {
      const number = streamNumber++;
      addToEventLog(`New incoming unidirectional stream #${number}`);
      readFromIncomingStream(readable, number);
    }
    addToEventLog('Done accepting unidirectional streams!');
  } catch (e) {
    addToEventLog(`Error while accepting streams ${e}`, 'error');
  }
}

async function readFromIncomingStream(readable, number) {
  try {
    const decoder = new TextDecoderStream('utf-8');
    for await (const data of readable.pipeThrough(decoder)) {
      addToEventLog(`Received data on stream #${number}: ${data}`);
    }
    addToEventLog(`Stream #${number} closed`);
  } catch (e) {
    addToEventLog(`Error while reading from stream #${number}: ${e}`, 'error');
    addToEventLog(`    ${e.message}`);
  }
}

function addToEventLog(text, severity = 'info') {
  const log = document.getElementById('event-log');
  const previous = log.lastElementChild;
  const entry = document.createElement('li');
  entry.innerText = text;
  entry.className = `log-${severity}`;
  log.appendChild(entry);

  // If the previous entry in the log was visible, scroll to the new element.
  if (previous &&
      previous.getBoundingClientRect().top < log.getBoundingClientRect().bottom) {
    entry.scrollIntoView();
  }
}
```

§ 16. Acknowledgements

The editors wish to thank the Working Group chairs and Team Contact, Jan-Ivar Bruaroey, Will Law and Yves Lafon, for their support.

The `WebTransport` interface is based on the `QuicTransport` interface initially described in the W3C ORTC CG, and has been adapted for use in this specification.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[] defines the following terms:
- cross-site recognition

[CSP3] defines the following terms:
- Report Content Security Policy violations for request
- Should request be blocked by Content Security Policy?
- webrtc

[DOM] defines the following terms:
- abort reason
- add

[ECMASCRIPT-6.0] defines the following terms:
- fulfilled
- rejected
- resolved
- settled
- the typed array constructors table

[ENCODING] defines the following terms:
- TextDecoderStream
- TextEncoderStream
- UTF-8 decode
- UTF-8 encode

[FETCH] defines the following terms:

- block bad port
- client
- connection
- credentials
- destination
- determine the network partition key
- fetch
- obtain a connection
- origin
- origin (for request)
- policy container
- redirect mode
- request
- stream
- URL

[HR-TIME-3] defines the following terms:

- DOMHighResTimeStamp

[HTML] defines the following terms:

- Serializable
- Transferable
- API base URL
- ASCII serialization of an origin
- deserialization steps
- event loop
- in parallel
- networking task source
- origin
- origin (for environment settings object)
- policy container
- queue a global task
- relevant global object
- relevant settings object
- serializable object
- serialization steps
- transfer steps
- transfer-receiving steps

[INFRA] defines the following terms:

- abort when
- append
- ASCII case-insensitive
- break
- byte sequence
- code unit prefix
- contain
- dequeue
- empty
- enqueue
- exist
- for each
- implementation-defined
- isomorphic encode
- length
- ordered set
- queue
- remove
- set

[STREAMS] defines the following terms:

- ReadableStream
- WritableStream
- WritableStreamDefaultWriter
- abort() (for WritableStream)
- abort() (for WritableStreamDefaultWriter)
- abortAlgorithm
- BYOB reader
- cancel() (for ReadableStream)
- cancel() (for ReadableStreamGenericReader)
- cancelAlgorithm
- close (for ReadableStream)
- close (for WritableStream)
- close() (for WritableStream)
- close() (for WritableStreamDefaultWriter)
- closeAlgorithm
- current BYOB request view
- default reader
- enqueue
- error (for ReadableStream)
- error (for WritableStream)
- getWriter()
- high water mark
- highWaterMark
- locked
- pull from bytes
- pullAlgorithm (for ReadableStream/set up)
- pullAlgorithm (for ReadableStream/set up with byte reading support)
- read()
- readable byte stream
- ready
- set up (for ReadableStream)
- set up (for WritableStream)
- set up with byte reading support
- write()
- write(chunk)
- writeAlgorithm

[URL] defines the following terms:

- fragment
- scheme
- URL parser
- URL record

[WEBIDL] defines the following terms:

- AbortError
- ArrayBuffer
- BufferSource
- Clamp
- DOMException
- DOMString
- DataView
- EnforceRange
- Exposed
- InvalidStateError
- NewObject
- NotSupportedError
- Promise
- QuotaExceededError
- RangeError
- SecureContext
- SyntaxError
- TypeError
- USVString
- Uint8Array
- a promise rejected with
- a promise resolved with
- any
- boolean
- byte length
- code
- created
- long long
- message
- name
- new
- reacting
- sequence
- this
- throw
- undefined
- underlying buffer
- unrestricted double
- unsigned long
- unsigned long long
- unsigned short
- upon fulfillment
- write

[WEBTRANSPORT] defines the following terms:

- [[OutgoingDatagramsQueue]]

§ References

§ Normative References

[CSP3]
Mike West; Antonio Sartori. *Content Security Policy Level 3*. URL: https://w3c.github.io/webappsec-csp/

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMASCRIPT-6.0]
Allen Wirfs-Brock. *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification*. URL: http://www.ecma-international.org/ecma-262/6.0/index.html

[ENCODING]
Anne van Kesteren. *Encoding Standard*. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]
Anne van Kesteren. *Fetch Standard*. Living Standard. URL: https://fetch.spec.whatwg.org/

[HR-TIME-3]
Yoav Weiss. *High Resolution Time*. URL: https://w3c.github.io/hr-time/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[QUIC]
Jana Iyengar; Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9000

[QUIC-DATAGRAM]
Tommy Pauly; Eric Kinnear; David Schinazi. *An Unreliable Datagram Extension to QUIC*. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9221

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[RFC3279]
L. Bassham; W. Polk; R. Housley. *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. April 2002. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc3279

[RFC5280]
D. Cooper; et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc5280

[RFC8174]
B. Leiba. *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[RFC8422]
Y. Nir; S. Josefsson; M. Pegourie-Gonnard. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*. August 2018. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8422

[RFC9002]
J. Iyengar, Ed.; I. Swett, Ed.. *QUIC Loss Detection and Congestion Control*. May 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9002

[RFC9525]
P. Saint-Andre; R. Salz. *Service Identity in TLS*. November 2023. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9525

[STREAMS]
Adam Rice; et al. *Streams Standard*. Living Standard. URL: https://streams.spec.whatwg.org/

[URL]
Anne van Kesteren. *URL Standard*. Living Standard. URL: https://url.spec.whatwg.org/

[WEB-TRANSPORT-HTTP2]
Alan Frindell; et al. *WebTransport over HTTP/2*. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http2/

[WEB-TRANSPORT-HTTP3]
Alan Frindell; Eric Kinnear; Victor Vasiliev. *WebTransport over HTTP/3*. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3/

[WEB-TRANSPORT-OVERVIEW]
Victor Vasiliev. *WebTransport Protocol Framework*. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-overview

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

[WEBTRANSPORT]
Nidhi Jaju; Victor Vasiliev. *WebTransport*. URL: https://w3c.github.io/webtransport/

§ Informative References

[RFC7301]
S. Friedl; et al. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*. July 2014. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7301

[RFC8446]
E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. August 2018. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8446

[RFC9308]
M. Kühlewind; B. Trammell. *Applicability of the QUIC Transport Protocol*. September 2022. Informational. URL: https://www.rfc-editor.org/rfc/rfc9308

[UNSANCTIONED-TRACKING]
Mark Nottingham. *Unsanctioned Web Tracking*. 17 July 2015. TAG Finding. URL: http://www.w3.org/2001/tag/doc/unsanctioned-tracking/

[WEBRTC]
Cullen Jennings; et al. *WebRTC: Real-Time Communication in Browsers*. URL: https://w3c.github.io/webrtc-pc/

§ IDL Index

```
[Exposed=(Window,Worker), SecureContext, Transferable]
interface WebTransportDatagramsWritable : WritableStream {
  attribute WebTransportSendGroup? sendGroup;
  attribute long long sendOrder;
};
```

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransportDatagramDuplexStream {
  WebTransportDatagramsWritable createWritable(
      optional WebTransportSendOptions options = {});
  readonly attribute ReadableStream readable;

  readonly attribute unsigned long maxDatagramSize;
  attribute unrestricted double? incomingMaxAge;
  attribute unrestricted double? outgoingMaxAge;
  attribute unrestricted double incomingHighWaterMark;
  attribute unrestricted double outgoingHighWaterMark;
};

[Exposed=(Window,Worker), SecureContext]
interface WebTransport {
  constructor(USVString url, optional WebTransportOptions options = {});

  Promise<WebTransportConnectionStats> getStats();
  [NewObject] Promise<ArrayBuffer> exportKeyingMaterial(BufferSource label, optional Buffe
  readonly attribute Promise<undefined> ready;
  readonly attribute WebTransportReliabilityMode reliability;
  readonly attribute WebTransportCongestionControl congestionControl;
  [EnforceRange] attribute unsigned short? anticipatedConcurrentIncomingUnidirectionalStre
  [EnforceRange] attribute unsigned short? anticipatedConcurrentIncomingBidirectionalStrea
  readonly attribute DOMString protocol;

  readonly attribute Promise<WebTransportCloseInfo> closed;
  readonly attribute Promise<undefined> draining;
  undefined close(optional WebTransportCloseInfo closeInfo = {});

  readonly attribute WebTransportDatagramDuplexStream datagrams;

  Promise<WebTransportBidirectionalStream> createBidirectionalStream(
      optional WebTransportSendStreamOptions options = {});
  /* a ReadableStream of WebTransportBidirectionalStream objects */
  readonly attribute ReadableStream incomingBidirectionalStreams;

  Promise<WebTransportSendStream> createUnidirectionalStream(
      optional WebTransportSendStreamOptions options = {});
  /* a ReadableStream of WebTransportReceiveStream objects */
  readonly attribute ReadableStream incomingUnidirectionalStreams;
  WebTransportSendGroup createSendGroup();

  static readonly attribute boolean supportsReliableOnly;
};

enum WebTransportReliabilityMode {
  "pending",
  "reliable-only",
  "supports-unreliable",
};
```

```
dictionary WebTransportHash {
  DOMString algorithm;
  BufferSource value;
};

dictionary WebTransportOptions {
  boolean allowPooling = false;
  boolean requireUnreliable = false;
  sequence<WebTransportHash> serverCertificateHashes;
  WebTransportCongestionControl congestionControl = "default";
  [EnforceRange] unsigned short? anticipatedConcurrentIncomingUnidirectionalStreams = null
  [EnforceRange] unsigned short? anticipatedConcurrentIncomingBidirectionalStreams = null;
  sequence<DOMString> protocols = [];
};

enum WebTransportCongestionControl {
  "default",
  "throughput",
  "low-latency",
};

dictionary WebTransportCloseInfo {
  unsigned long closeCode = 0;
  USVString reason = "";
};

dictionary WebTransportSendOptions {
  WebTransportSendGroup? sendGroup = null;
  long long sendOrder = 0;
};

dictionary WebTransportSendStreamOptions : WebTransportSendOptions {
  boolean waitUntilAvailable = false;
};

dictionary WebTransportConnectionStats {
  unsigned long long bytesSent = 0;
  unsigned long long packetsSent = 0;
  unsigned long long bytesLost = 0;
  unsigned long long packetsLost = 0;
  unsigned long long bytesReceived = 0;
  unsigned long long packetsReceived = 0;
  required DOMHighResTimeStamp smoothedRtt;
  required DOMHighResTimeStamp rttVariation;
  required DOMHighResTimeStamp minRtt;
  required WebTransportDatagramStats datagrams;
  unsigned long long? estimatedSendRate = null;
  boolean atSendCapacity = false;
};

dictionary WebTransportDatagramStats {
  unsigned long long droppedIncoming = 0;
  unsigned long long expiredIncoming = 0;
  unsigned long long expiredOutgoing = 0;
  unsigned long long lostOutgoing = 0;
};

[Exposed=(Window,Worker), SecureContext, Transferable]
interface WebTransportSendStream : WritableStream {
  attribute WebTransportSendGroup? sendGroup;
  attribute long long sendOrder;
  Promise<WebTransportSendStreamStats> getStats();
  WebTransportWriter getWriter();
};

dictionary WebTransportSendStreamStats {
  unsigned long long bytesWritten = 0;
  unsigned long long bytesSent = 0;
  unsigned long long bytesAcknowledged = 0;
};

[Exposed=(Window,Worker), SecureContext]
interface WebTransportSendGroup {
  Promise<WebTransportSendStreamStats> getStats();
};

[Exposed=(Window,Worker), SecureContext, Transferable]
interface WebTransportReceiveStream : ReadableStream {
  Promise<WebTransportReceiveStreamStats> getStats();
};

dictionary WebTransportReceiveStreamStats {
  unsigned long long bytesReceived = 0;
  unsigned long long bytesRead = 0;
};

[Exposed=(Window,Worker), SecureContext]
interface WebTransportBidirectionalStream {
  readonly attribute WebTransportReceiveStream readable;
  readonly attribute WebTransportSendStream writable;
};

[Exposed=*, SecureContext]
interface WebTransportWriter : WritableStreamDefaultWriter {
  Promise<undefined> atomicWrite(optional any chunk);
};

[Exposed=(Window,Worker), Serializable, SecureContext]
interface WebTransportError : DOMException {
  constructor(optional DOMString message = "", optional WebTransportErrorOptions options =

  readonly attribute WebTransportErrorSource source;
  readonly attribute unsigned long? streamErrorCode;
};

dictionary WebTransportErrorOptions {
  WebTransportErrorSource source = "stream";
  [Clamp] unsigned long? streamErrorCode = null;
};

enum WebTransportErrorSource {
  "stream",
  "session",
};
```

§ Issues Index

ISSUE 1 This needs to be done in workers too. See #127 and whatwg/html#6731. ↵

ISSUE 2
This configuration option is considered a feature at risk due to the lack of implementation in browsers of a congestion control algorithm, at the time of writing, that optimizes for low latency.
↵