RTCQuicTransport & RTCIceTransport Chrome Design Doc (public version)

This Document is Public

Authors: shampson@chromium.org, steveanton@chromium.org

One-page overview

Summary

WebRTC (NV) is moving towards lower level APIs and as part of this effort there are extension web specifications for both a RTCQuicTransport & a RTCIceTransport (work in progress editor's drafts). The RTCIceTransport is used to establish peer-to-peer connections using the ICE protocol and is of little use on its own. The RTCQuicTransport built with a RTCIceTransport provide Web developers with a generic data transport using the QUIC protocol.

For more information please refer to the Explainer Doc.

Platforms

Blink platforms: Desktop + Chrome OS + Android + Fuchsia

Team

shampson@chromium.org steveanton@chromium.org
TL/manager: emadomara@chromium.org

Bug

Launch bug:
https://bugs.chromium.org/p/chromium/issues/detail?id=868068

Code affected

Blink:
The development will be done in the same blink location as other WebRTC blink Web APIs - //src/third_party/blink/renderer/modules/peerconnection.

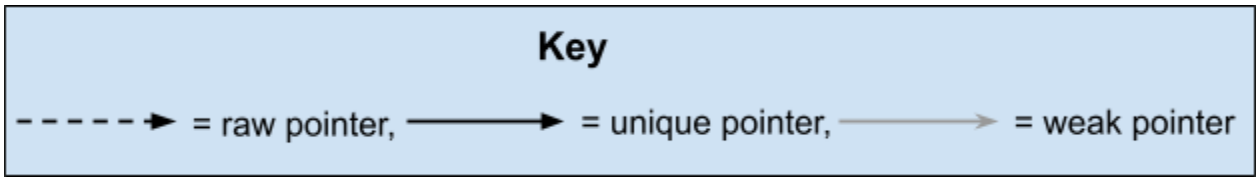
Quic:
We will need to use the core quic library (//src/net/third_party/quic/core) that exists in Chromium's third_party directory. This requires subclassing and using APIs that are currently marked QUIC_EXPORT_PRIVATE from the blink directory (we'll need to change these exports to public).

Design

The underlying networking engines are already implemented as libraries and will be re-used by the Web API bindings. The ICE engine is implemented in WebRTC and exposed via the p2ptransportchannel.h . In the future this will likely change to be exposed via an IceInterface, but for now we will use the P2PTransportChannel directly. The QUIC engine is implemented in the QUIC library and exposed via the QuicStream & QuicSession (which clients should subclass). These subclassed objects (BlinkQuicSession/BlinkQuicStream) will exist in blink.

To avoid the main JavaScript thread blocking network processing and vice-versa the ICE and QUIC networking is run on a separate thread within the renderer process. This is the WebRTC worker thread, and will be referenced as the “network thread” in this document.

Note: In the following diagrams pointers are represented by:

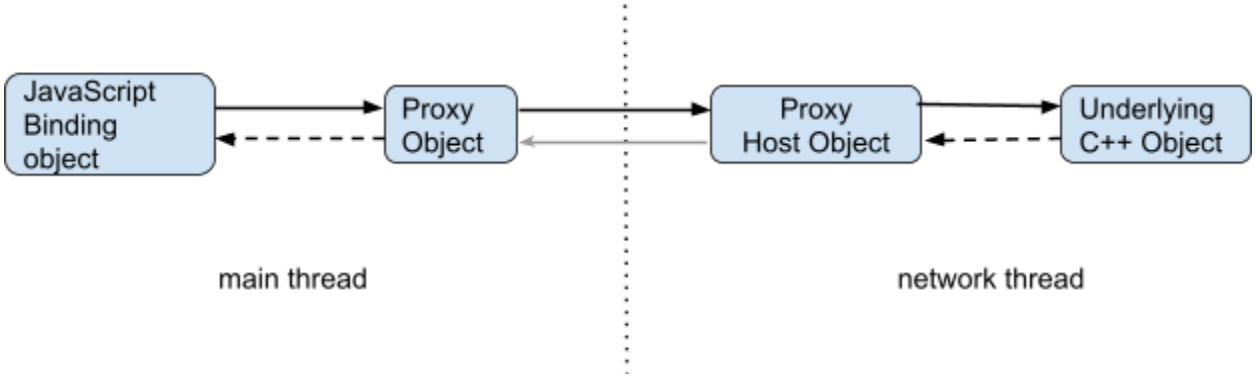


Cross-Thread Object Interaction

The JavaScript bindings need to issue control commands to the networking engines running on the network thread. Likewise, the bindings need to receive event callbacks originating from the network thread. The actual packet flow between the underlying ICE and QUIC transports does not need to leave the network thread.

To ensure consistency in the Web API implementation, all interactions between the threads should happen asynchronously using PostTask with synchronous operations relying on state cached in the binding object. This ensures that calls from JavaScript to the binding object are processed only using information that has already been returned via event callbacks.

A general design that satisfies this constraint is shown below:



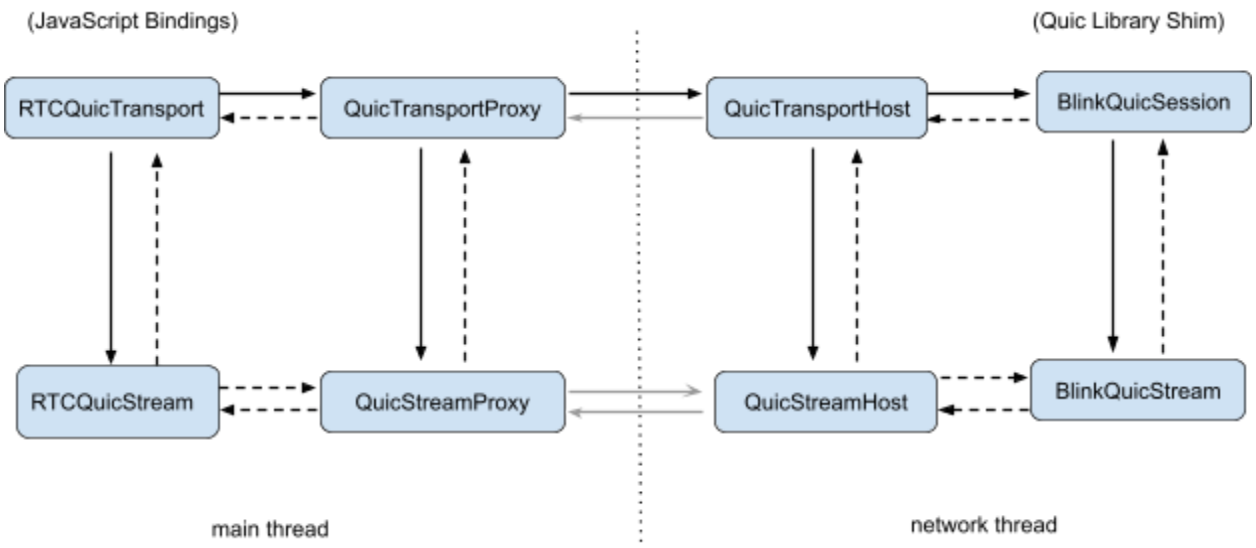
The JavaScript (Blink binding) object is garbage collected, while the Proxy/Proxy Host objects are not. The Proxy object is owned by the Blink binding and exposes a control surface similar to the Web API. These methods are implemented by posting a task to the Proxy Host object which calls the appropriate method on the underlying C++ object (e.g., P2PTransportChannel, QuicSession). Callbacks are posted using the reverse pointers. This keeps the cross-thread details hidden inside the Proxy/Host objects. All of these objects will live in the blink directory.

Quic Object Ownership

The QUIC object ownership model is somewhat complicated due to the transient nature of QUIC streams combined with the multi-threading requirements. The design must handle the following cases:

- Creating a new QUIC stream from the Web API.
- Creating a new Web API stream when the QUIC library indicates the remote side has created a new stream.
- Closing a QUIC stream from the Web API.
- Closing a Web API stream when the QUIC library indicates the remote side has closed the stream.
- Stopping the QUIC connection from the Web API.
- Stopping the Web API objects when the QUIC library indicates that the remote side has closed the connection.

The following object model satisfies these requirements:



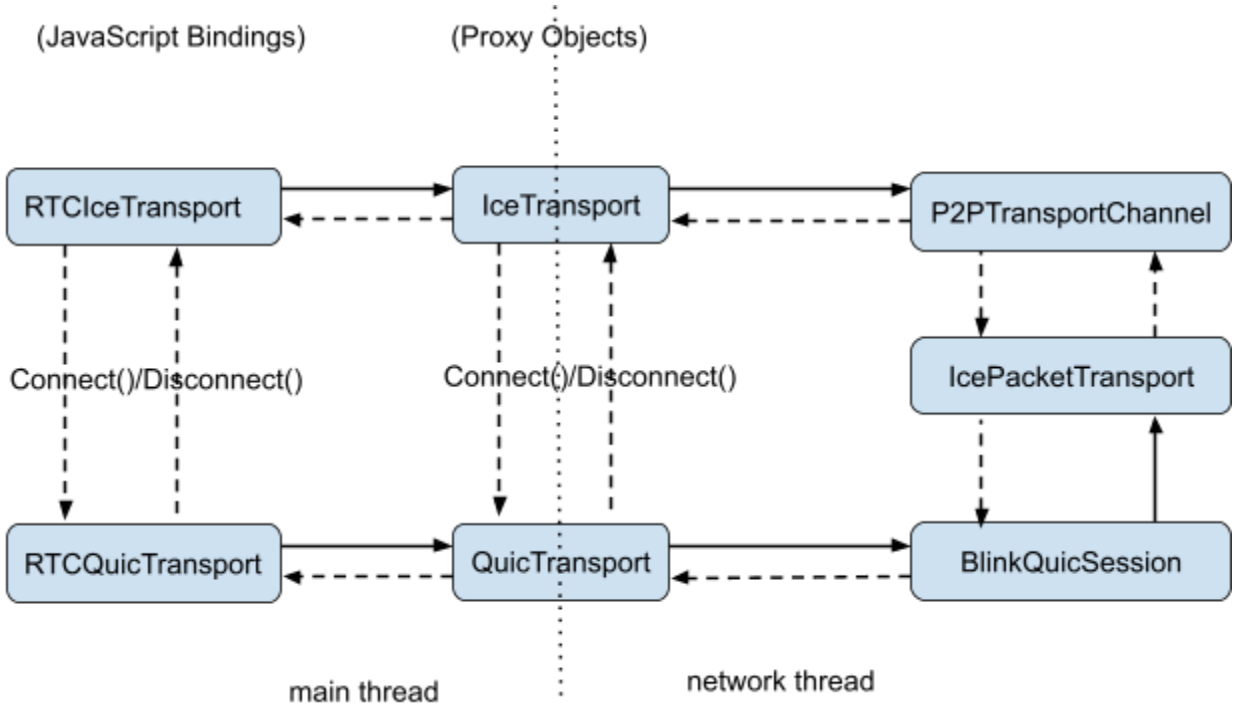
Importantly, the ownership model of the Proxy/Host objects mirrors that of the core QUIC library objects (QuicSession owning the QuicStream), reducing cognitive load. Objects can be created and destroyed from either thread and weak pointers ensure that in-flight tasks are silently dropped if the target object is deleted. By having the QuicTransport own all the QuicStream objects, the implementation of RTCQuicTransport.stop() is simple since it just needs to post one task to delete all the underlying transport/stream objects. All of these objects will live in the blink directory, with the BlinkQuicSession & BlinkQuicStream subclassing the the core QUIC library's QuicSession and QuicStream.

For more detailed use cases of stream object creation/destruction please refer to this document: RTCQuicTransport & RTCQuicStream Creation/Destruction.

Ice Integration

Packets should flow between the underlying P2PTransportChannel and BlinkQuicSession without involvement from the main thread -- but the connection between these objects is controlled from the Web API. The BlinkQuicSession is designed to take a pointer to the underlying packet transport (P2PTransportChannel) and expects that packet transport to outlive it. There also needs to be a reference back from the P2PTransportChannel to the QuicTransport in order to receive incoming packets.

At all levels of QuicTransports (RTCQuicTransport, proxy & BlinkQuicSession) the connection to the same level IceTransport (RTCIceTransport, proxy, & P2PTransportChannel) is made in the constructor and is disconnected in the destructor. To ensure that a programmer error does not cause the IceTransport to be deleted prematurely, the IceTransport destructor DCHECKs that the transport using the IceTransport (in this case the QuicTransport) has already been disconnected.



Buffering Data

Since JavaScript is not real-time and cannot run at infinite speed, there needs to be a way to apply back pressure to the sender so that the receive buffer does not grow unbounded. At the same time, we want to ensure that the JavaScript does not get many callbacks for small bits of data when it wants to receive large chunks. The Web API has a mechanism to support this (waitForReadable, waitForWriteable), and we can see two ways to implement it:

Option A: Pass received data continuously to the main thread and post back on each readInto()

In this design, the RTCQuicStream object will get a task posted to it every time the underlying BlinkQuicStream receives data. The Blink binding will store an internal buffer of the data and be responsible for resolving promises and handling reads. Back pressure to the sender is managed by the QuicStreamHost tracking the RTCQuicStream's buffer size. It will not read and consume more data once this buffer size reaches the target read buffered amount. When a readInto() occurs, a task will be posted to the QuicStreamHost to update its view of the buffer size.

Option B: Use a shared buffer and snapshot the buffer size on the first readInto() call

This design has the main thread and network thread sharing the receive buffer and waitForReadable marks using a mutex to guard access. The main thread will need to snapshot the size of the receive buffer at the first call to read() so that it does not expose packets that have arrived after the task started.

When data is received from the QUIC library, append to the shared buffer if the buffer size would not be exceeded. If one of the waitForReadable promises should be resolved, post a task to the main thread to resolve it. A readInto() will return data from the shared buffer, increasing the buffer size immediately.

Evaluation of Options:

Option A is simpler to implement because the buffer is not shared and the JavaScript bindings handle promise resolutions. Although, it may not be as high performing because it will involve posting many more tasks for high throughput data transfers. Option B may be higher performing since it does not involve as many posted tasks on reads (only when a waitForReadablePromise should be resolved) but is more will likely be more complicated to implement correctly.

Metrics

Success metrics

We'll gauge success by measuring usage of the new Web APIs with Blink usage counters.

Regression metrics

These are new Web APIs so there won't be any regressions over expected behavior for Web pages using these APIs.

We can watch the speed launch metrics for regressions when a page is loaded that uses the new APIs.

Experiments

We do not plan to run any Finch experiments.

Rollout plan

Development will happen behind a Blink feature flag and follow the dev-beta-stable progression.

We may first experiment with an Origin trial. Otherwise, we'll rollout by sending a Blink Intent to Ship and enabling the feature flag by default.

Core principle considerations

Everything we do should be aligned with and consider Chrome's core principles. If there are any specific stability concerns, be sure to address them with appropriate experiments.

Speed

We expect the RTCQuicTransport will have a similar performance profile as the current RTCDataChannel (implemented with SCTP).

Since this is a new Web API that is not a drop-in replacement for the current RTCDataChannel, it will not be possible to A/B test this with a Finch experiment.

Regardless, we can watch for regressions in the speed launch metrics for browsers that load a page using the RTCQuicTransport.

Security

General WebRTC security considerations are documented in the WebRTC Security Architecture document.

More specific security considerations are called out in the WebRTC-ICE and WebRTC-QUIC specifications.

Network Access

Most of WebRTC (including these new APIs) runs inside the renderer process. Networking is routed to the browser process through P2P IPC calls (renderer, host). The P2P IPC host verifies that ICE consent is given by the peer before allowing the renderer to send or receive arbitrary data. The QUIC protocol will run on top of ICE in these APIs so the existing WebRTC security properties will apply.

Crypto Handshake

WebRTC has a special cryptographic handshake to handle peer-to-peer connections. A certificate authority can't be used to validate certificates because we have dynamic IPs (more information in section 3.3 rfc 8122). Instead, self signed certificates are used and the fingerprints are signaled over a secure signaling channel. The crypto handshake verifies that the fingerprint of the certificate used matches the fingerprint that was signalled. This is currently how WebRTC's PeerConnection establishes a secure DTLS connection (rfc 5763 section 5) and the RTCQuicTransport would not introduce any new security problems here.

Although, the QUIC library in Chromium currently only allows validation of the server's certificate. This will continue to be the case until QUIC supports the TLS 1.3 handshake, which is part of the team's future plans. If we expose the API through an origin trial before this is supported we have several options:

- Enforce server side certificate verification. This is suboptimal, because it does not verify both side's certificates for the P2P case the API is designed for.
- Signal a pre shared key out of band to be used with QUIC (spec issue here). This gives us both side security, but similar to SDES, exposes the key to JavaScript.
- Do a DTLS handshake using WebRTC's DtlsTransport to negotiate a symmetric key. This allows us to verify both certificates by using the signaled remote certificate fingerprints (as intended in the RTCQuicTransport and specified in DTLS-SRTP). This would give the full security guarantees, but requires any server to implement the DTLS handshake that wants to interact with an RTCQuicTransport browser endpoint.

We do not plan to ship the API until TLS 1.3 is supported.

Fuzzing

The ICE protocol uses STUN messages which are already covered by a WebRTC fuzzer.

The QUIC protocol is covered by fuzzers maintained alongside the QUIC code.

Do we need a fuzzer for the RTCQuicStream read/write API?

Privacy considerations

General WebRTC privacy considerations are documented in the WebRTC Security Architecture document.

The RTCIceTransport API does not expose anything beyond what is already possible with the RTCPeerConnection API. Permissions are not requested for use of a RTCDataChannel, and the plan is the same for RTCIceTransport/RTCQuicTransport.

The RTCQuicTransport API is not expected to put the user at any additional privacy risk.

Testing plan

We do not expect any additional testing to be needed beyond the usual waterfall tests, Web platform tests and unit tests.

Follow-up work

We will assess success by monitoring usage of the new Web APIs and soliciting feedback from Web developers.

Once the feature has been shipped and is considered stable, the Blink feature flag can be removed.