Flow control in QUIC

Robbie Shade (rjshade@google.com)
May, 2016

Overview

QUIC provides both stream and connection level flow control, similar to HTTP2. An endpoint sends WINDOW_UPDATE frames to the peer to increase flow control window size, and BLOCKED frames to indicate it has data to write but is blocked by flow control.

A motivating example is a server with limited memory: it needs a mechanism by which it can limit the amount of data each client can send it, to avoid allocating arbitrarily large buffers.

Stream flow control

Stream flow control is the mechanism by which one endpoint of a QUIC connection informs the other end about how much data it is willing to receive on each stream.

In QUIC, flow control works by advertising the absolute byte offset in the stream which an endpoint is willing to receive. For example, if we tell the peer that they can send up to byte 200 on stream N, and they have already sent 150 bytes on that stream they may only write another 50 bytes before blocking. As we free up memory by consuming data from the stream, we will send a WINDOW_UPDATE frame allowing the peer to send more data.

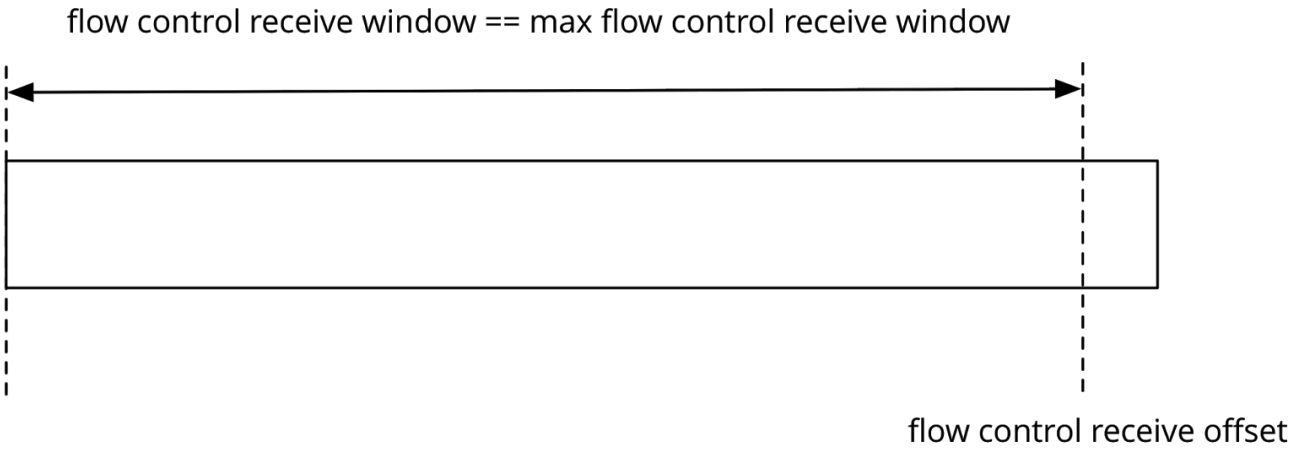

Figure (a): Empty stream

There is a maximum flow control receive window size which is chosen independently by each endpoint. An endpoint should base this decision on available resources - if you are not constrained by memory then set a large window of 100 Mb for example, whereas a busy server may set a window of 64 Kb. Figure (a) shows an empty stream after creation. It has not received any data from the peer, so its flow control receive window is equal to the maximum stream receive window. As data arrives, this receive window will shrink - eventually reaching 0 if the peer sends exactly enough data to fill it.

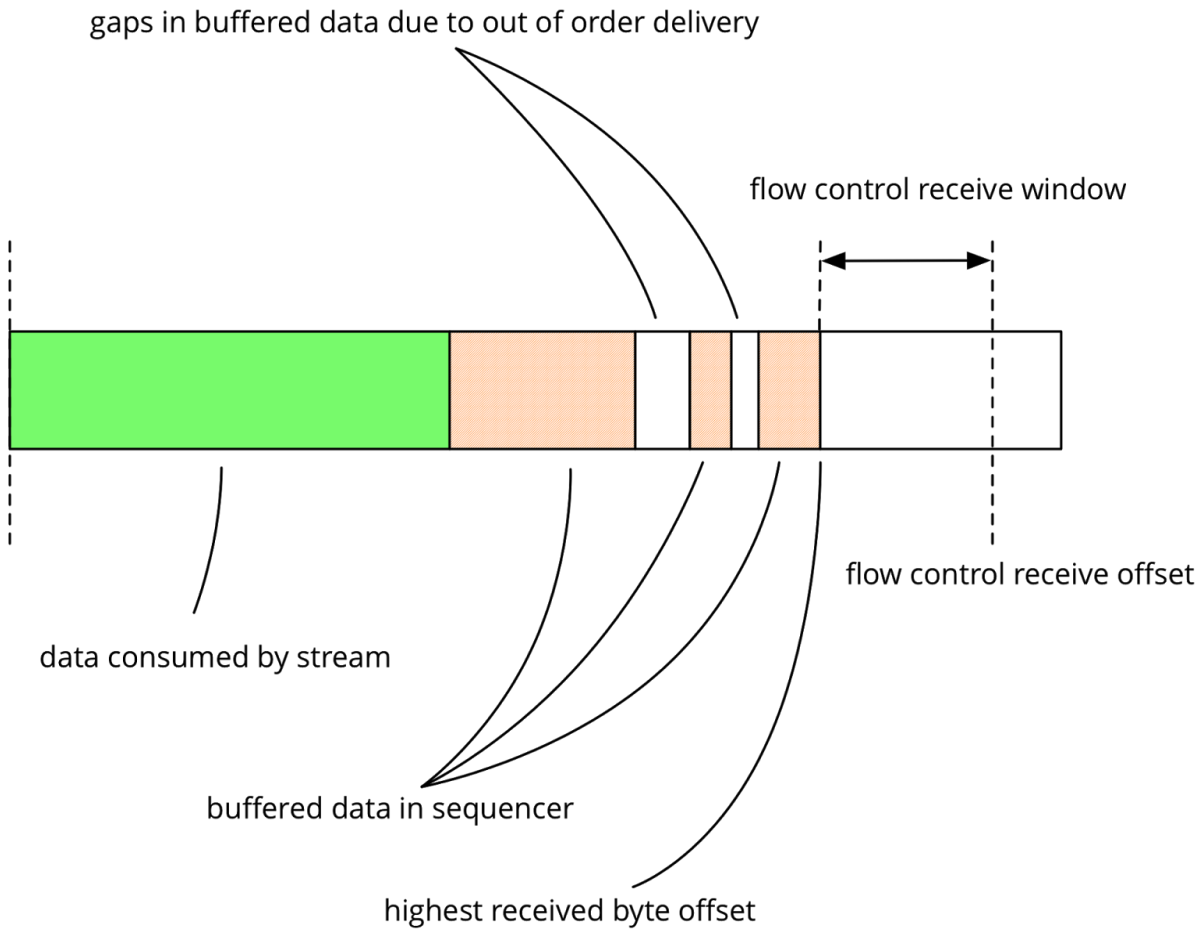

Figure (b): Stream has consumed some data, some buffered

This picture shows the situation after the peer has sent some data. We have consumed some bytes, and buffered some more. There are gaps in the buffered data as packets have arrived out of order. Notice that the flow control receive window has shrunk based on the **highest received byte offset** and not the amount of bytes buffered. This is reasonable as the peer does not send stream data with gaps: if there are gaps below the highest received offset this must be because packets have been reordered or dropped in the network.

We keep track of the highest byte offset we have seen on each stream. As new frames arrive we check to see if we need to change this offset, but it only ever increases. When a stream terminates the peer **must** inform us of the highest byte offset they have written to the stream. They can do this by either sending a data frame with a FIN, or a RST stream frame which contains a final byte offset.

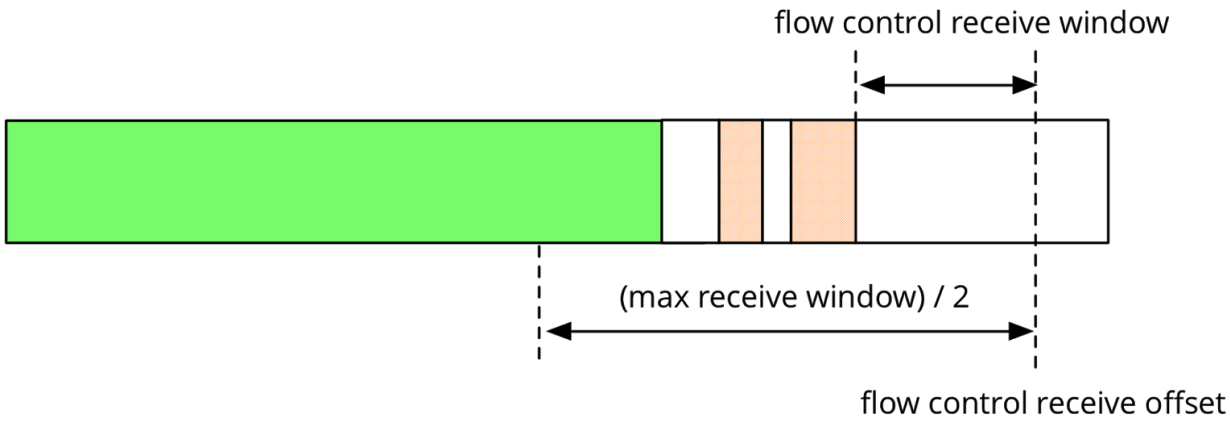

Figure (c): Stream has consumed enough data to trigger WINDOW_UPDATE

Now we get a chance to consume some more data from the stream. The heuristic used to determine when to send a WINDOW_UPDATE is the same as that used in [Chromium's HTTP2 implementation]:

```
 bool send_update =
    (flow control receive offset - consumed bytes) < (max receive window /
2);
```

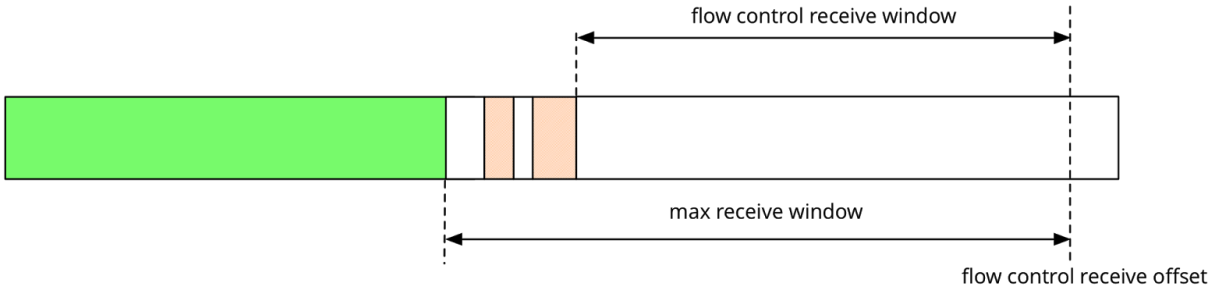

Figure (d): New receive window

We send a WINDOW_UPDATE pushing out the flow control receive offset such that it = (consumed bytes + max receive window).

Connection flow control

On its own, stream flow control is not sufficient to protect the receiver from too much incoming data. A client could open up to max_streams streams (currently 100) and send up to the per-stream flow control limit bytes on each stream. Therefore, in addition to limiting data sent on individual streams, we also require that the QUIC connection (potentially containing many streams) is itself flow controlled.

Connection flow control works in the same way as stream flow control, but the bytes consumed, and highest received offset, are the aggregate across all streams. As data is consumed by a stream it updates both its own bytes consumed field, and also the connection level flow controller.

For example, consider three streams, each of which has received some amount of data, but that data has not yet been fully consumed by the application.

- Stream 1 has received 100 bytes, and the application has consumed 80 bytes.
- Stream 2 has 90 bytes, and the application has consumed 50 bytes.
- Stream 3 has received 110 bytes, and the application has consumed 100 bytes.

Aggregating these gives a connection level flow control state where the connection has received 300 bytes, and consumed 230 of them.

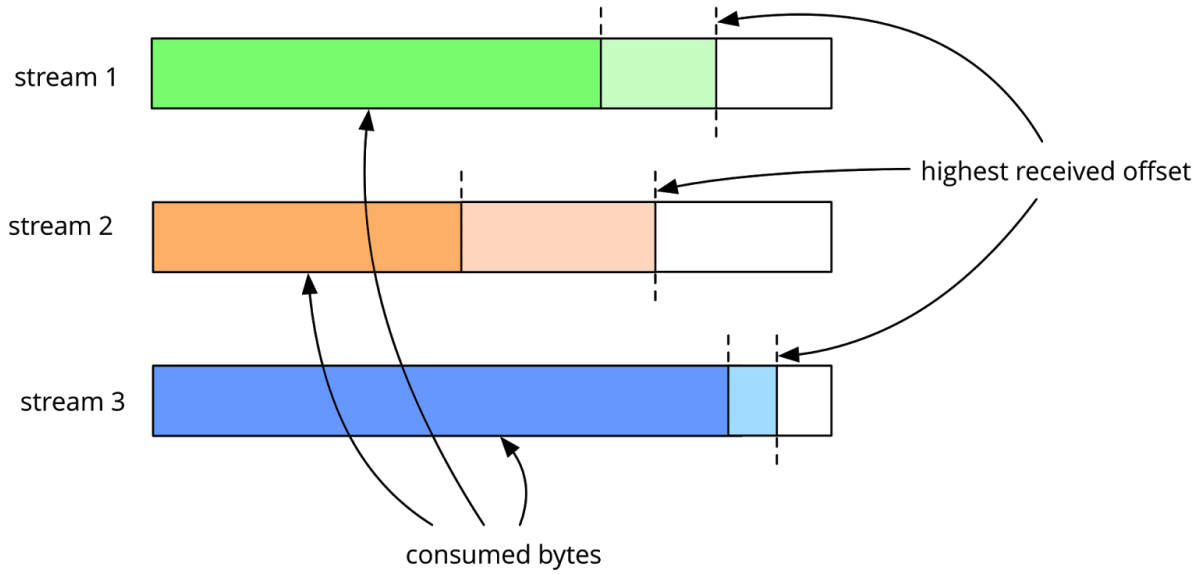

Figure (e): Individual streams with various offsets

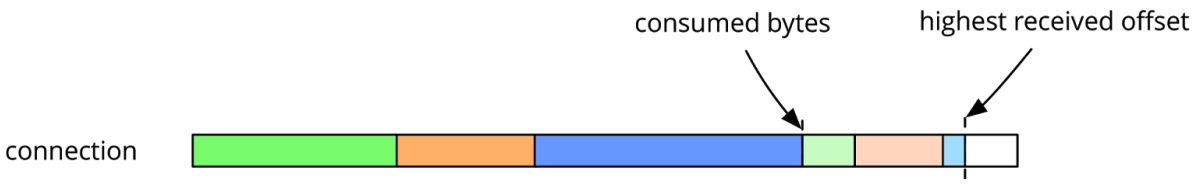

Figure (f): Connection level flow control aggregated across open streams

Having an overall connection level flow control window means that an individual slow stream (talking to a slow backend for example) won't completely starve the connection: WINDOW_UPDATEs at the connection level will allow other streams to progress, while the slow stream is blocked from receiving more data.

Other considerations and implementation details

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time (1 RTT hopefully?), both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive the connection will be torn down when too much data arrives. Conversely if endpoint A believes it is limited to sending X bytes, while endpoint B expects up to Y bytes (and Y > X), then we can get stuck in the situation where A is stalled on writing, waiting for a WINDOW_UPDATE which will never come.

RST mid-stream

What happens to connection level flow control when one side decides to reset a stream before it has finished sending/receiving data (e.g. the user closes a tab in Chrome)? On receipt of a RST frame, endpoint A will tear down the state for the matching stream, and ignore further data frames on that stream. This could potentially result in the endpoints getting out of sync: the RST may have arrived out of order (this is not TCP!) and there may be further bytes in flight (which B has counted against its connection level send window as they were sent).

Unless A can figure out how many bytes B sent on the stream it can't make the same adjustment in its flow controller receive window and the endpoint states will drift apart. To solve this, we include the final byte offset sent on the stream in every RST frame. This means that receiving a RST frame tells an endpoint definitively how many bytes the peer sent on that stream, and we can adjust our connection level window appropriately.

RST in reply to RST

B also needs to know how many bytes A sent on the stream which has been reset. A can't just tear down stream state on receipt of a RST - it also needs to tell B how many bytes it sent. There are two ways that an endpoint can definitively inform the peer of the number of bytes sent on a stream:

- RST frame with final byte offset
- Data frame with FIN bit set

Therefore the protocol requires that on stream termination each endpoint **must** send *either* a RST *or* a data frame with FIN. If you receive a RST and have sent neither a FIN nor a RST, you send a RST in response. You can then rest assured that the connection will deliver this to the peer, that you have the final byte offset *sent* by the peer, and you can adjust your connection level flow control state before tearing down the stream locally.

BLOCKED frames

A QUIC endpoint will send a BLOCKED frame (as described in the wire spec) if it has data to send but is currently flow control blocked. Ideally these will be sent very infrequently

as the endpoints adjust their windows via WINDOW_UPDATE to accommodate the current data rate, but BLOCKED frames have been invaluable for debugging and monitoring purposes.

Default values

At the time of writing, QUIC has the following default flow control values (from Chromium's quic_protocol.h):

```
// Minimum size of initial flow control window, for both stream and
session.
const uint32_t kMinimumFlowControlSendWindow = 16 * 1024;  // 16 KB

// Maximum flow control receive window limits for connection and stream.
const QuicByteCount kStreamReceiveWindowLimit = 16 * 1024 * 1024;   // 16
MB
const QuicByteCount kSessionReceiveWindowLimit = 24 * 1024 * 1024;  // 24
MB
```

The 16 KB minimum value is necessary to allow 0-RTT requests with bodies. Current implementations will advertise much larger windows during the handshake, via the kSFCW (Stream Flow Control Window) and kCFCW (Connection Flow Control Window) tags in the CHLO and SHLO.

Chromium currently sets:

```
    CFCW: 15728640 // 15 MB
    SFCW: 6291456  // 6 MB
```

Google's servers currently set:

```
    CFCW: 1572864 // 1.5 MB
    SFCW: 1048576 // 1 MB
```

Auto-tuning max receive window

Most TCP implementations allow the size of the receive buffer to be auto-tuned, e.g. see *tcp_moderate_rcvbuf* in the Linux tcp manpage. A similar scheme has been implemented for QUIC as follows.

Rationale

As with many of the TCP auto-tuning counterparts, the basic idea is to start with relatively small initial window size, and then grow the window as necessary. For simplicity, auto-tuning may increase the window size, but never decreases (contrast with congestion control).

The ideal size of the window is one that is large enough that it can encompass the bandwidth delay product (BDP) to the peer or the consuming application, whichever is less, but not much larger. Smaller window size than the ideal may harmful throughput, and larger would be wasteful.

As the window update is central to QUIC's flow controller design, it provides a very convenient point to implement an auto-tuning mechanism that finds the ideal window size. Specifically, the algorithm will compare the interval between successive flow control window updates to the (smoothed) RTT estimate already maintained by QUIC. If the flow control window is too small to keep up with the BDP, there will be a window update each RTT. Alternatively, when the window is sized to the ideal, window updates can be expected to occur with frequency corresponding to more than the 1 RTT indicative of blocking, but not too much more. The default target chosen for auto-tuning corresponds to 2 RTTs. (since updates are triggered at the half-way point of the window, this corresponds to a total window size target of 4 RTTs).

The algorithm also imposes an upper bound on window size, which may be set by as an additional safety check or in conjunction with explicit memory pressure response mechanisms.

Algorithm

- As above, the flow control window update triggered when:
  `available window < max window size / 2,`
  where `available window = max receive window offset - bytes consumed`
- to realize auto-tuning, add the following logic just before issuing a window update
  - keep track of time interval between subsequent window updates, call this `since_last_update.`
  - `if ( since_last_update < RTT * trigger factor ) then max window size = MIN(max window size * increase factor, upper bound).`
    - `trigger factor` is 2
    - `increase factor` is 2

- As above, window update sets:
  `max_received window offset += (max window size - available window)`

Default values

Google's servers default to enable auto-tuning for receive buffers, with the following initial window size settings:

```
CFCW: 49152 // 48 KB
SFCW: 32768 // 32 KB
```

Chromium defaults disable auto-tuning for its receive buffers.