Why

The transport becomes more generic, simpler

A litany of small issues:

#515 - the server can’t speak first

#281 - HOLB on server push

Client has to send empty STREAM+FIN for server push
Symmetrical Streams

Unidirectionality has an effect on protocols with 1-to-1 mappings between client and server messages

... which might seem like it applies to most protocols

... except HTTP server push, RTP, CoAP, maybe others

1-to-1 and streams-as-messages don’t always fit

The cost is that 1-to-1 protocols need an explicit correlator

... stream ID no longer works as implicit correlator
After a 1-RTT TLS 1.3 handshake, the server speaks first
In HTTP over QUIC, the server could speak first, but
stream 1 is a client-initiated stream
We could use stream 2, but
in 0-RTT the client speaks first
Using stream 1 for 0-RTT and stream 2 for 1-RTT is gross
Server push operates in two stages:

1. the promise stage where new “requests” are created
2. the fulfilment state where responses are sent

In practice, promises are often generated opportunistically

... usually as new resources are “discovered”

Order of promises doesn’t matter

... but fulfilment order is critical to performance
If a server promises more than the maximum stream number...

A resource that is “discovered” late can be stalled

If that resource is urgent, it might take an RTT to sort out

Ideally, promises could be fulfilled in any order

That requires a layer of indirection...
Current State Machine

** or any mention of a higher-numbered stream

- **Idle**
  - send/recv STREAM **

- **Open**
  - recv RST_STREAM
  - recv all data or RST_STREAM
  - send all data or RST_STREAM
  - send RST_STREAM

- **Read Closed**
  - send all data or RST_STREAM

- **Write Closed**
  - recv all data or RST_STREAM

- **Closed**
Current State Machine

Idle

- send/recv STREAM **
- Send RST_STREAM

Open

- recv RST_STREAM
- Send RST_STREAM
- send all data or RST_STREAM
- recv all data or RST_STREAM

Closed

- Send RST_STREAM
- recv all data or RST_STREAM
- Send RST_STREAM
- Send RST_STREAM

Read Closed

- send all data or RST_STREAM

Write Closed

- recv all data or RST_STREAM
Simplification

idle

open

closed

send STREAM **

recv all data or RST_STREAM

send all data or RST_STREAM

** or any mention of a higher-numbered stream
HTTP Impact
HTTP Mapping Strawman

Changes:

response streams include the stream ID of the request

push streams include the stream ID of the response

stream that contained the promise, and a promise index

\{ header streams have a flag if there is a body

body streams include the stream ID of the headers

\} need a new control frame to cancel a push

maybe see \#557
Advantages

No empty streams
No odds/evens for streams
  ... twice as many requests possible
  ... more when requests/responses have no body
Push fulfilment in any order
Endpoints can send SETTINGS immediately
Maybe-Negative Consequences

Server push is counted with responses against stream limit
server now has to decide which to answer

Extra correlators on the start of server streams
though probably less was used on empty streams
most will be relative, so we can find an efficient encoding
Non-Obvious Consequences

A client can make more requests than the server can answer.

With bidirectional streams, client MAX_STREAM_ID governs pushes, which have this exact problem.

Recommendation:

Don’t worry about it, let the server reset requests.

If it hurts, stop: advertise a larger limit.
Example (Bidirectional)

C

GET / (stream: 9)

“” (stream: 11)

200 OK (stream: 9, body: true)

PUSH_PROMISE GET /tracker.gif (stream: 9, promise: 18)

PUSH_PROMISE GET /index.js (stream: 9, promise 22)

“Hello World!” (stream: 11)

200 OK (stream: 22)

“window.alert(‘hi’);” (stream: 24)

204 (No Content) (stream: 18)

“” (stream: 20)

“” (stream: 22), “” (stream 24)

“” (stream: 18), “” (stream 20)

S
Example (Unidirectional)

GET / (stream: 8, body: false)

200 OK (stream: 17, body: true)

PUSH_PROMISE GET /tracker.gif (stream: 17, body: false)

PUSH_PROMISE GET /index.js (stream: 17)

“Hello World!” (stream: 18, response: 17)

200 OK (stream: 19, promise: 17:1)

“window.alert(‘hi’);” (stream: 20, response: 19)

204 (No Content) (stream: 21, promise: 17:0)