A QUIC future in Rust

Dirkjan Ochtman
RustFest Paris, May 2018

https://github.com/djc/quinn

About me

 **@djc**  **@djco**

- https://dirkjan.ochtman.nl/
- Rust projects: Askama, tokio-imap & imap-proto
- Contributed to: cargo, *ring*, rustls, planetrs, Gentoo Linux maintainer for rust/cargo ebuilds
- Currently in between jobs. Would love to help you or your employer fill out gaps in the Rust ecosystem!

Started Rust 2 years ago.

What is QUIC?

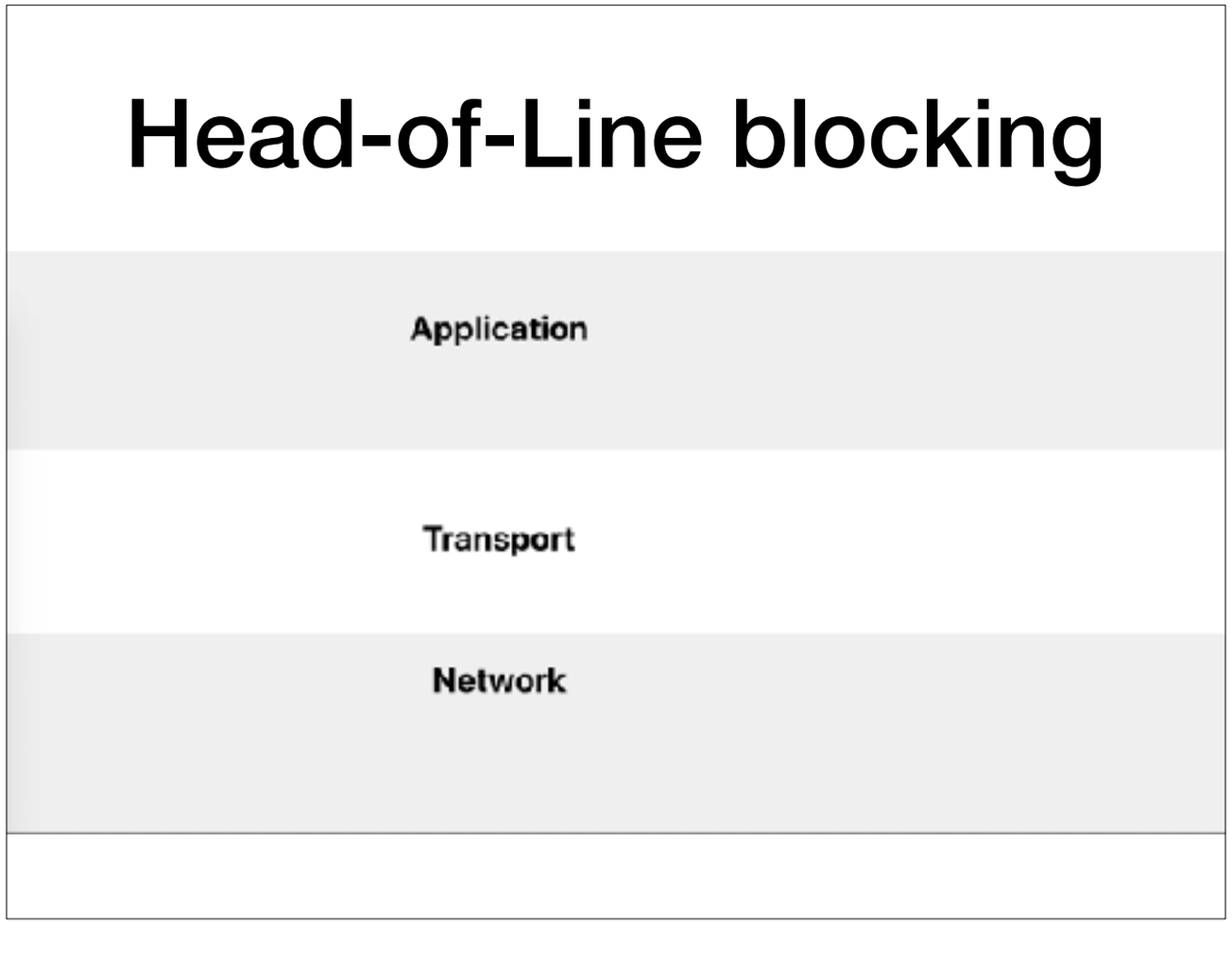
- Quick UDP Internet Connections
- Re-envision TCP after ~40 years of use
- Especially geared towards use as an HTTP (2) transport
- Started at Google in 2012, IETF working group established in 2016, planning for final drafts in Nov 2018
- (SPDY announced 2009; first draft of HTTP 2 in 2012; RFC 7540 published May 2015; SPDY removed in 2016.)

RFC 793 published in 1981.

Table of contents

- **QUIC: problem statement, concepts and process**
- Rust ecosystem: rustls, *ring*, futures, tokio, h2, bytes
- Quinn: design, implementation status, future

Head-of-Line blocking



QUIC goals

- Minimize latency
- Provide multiplexing
- Changes limited to path endpoints
- Enable multipath and forward error correction
- Provide always-secure transport
- Expect rapid distributed development and testing

Latency: connection establishment and overall transport for applications, starting with HTTP/2.

QUIC documents

- Invariants: things that cannot change in future versions
- Transport: core protocol logic
- Recovery: loss detection and congestion control
- TLS: how to secure the transport payloads
- QPACK: header compression (static & dynamic)
- HTTP: mapping of HTTP (2) semantics to QUIC

4 kinds of streams; stream 0 reserved for TLS
HTTP uses stream 2 and 3 for control traffic

Recovery

- Differences with TCP
  - Monotonically increasing packet numbers
  - No reneging (reduces memory pressure on sender)
  - More ACK ranges (less reliant on timeouts)
  - Explicit correction for delayed acknowledgements

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number. QUIC separates the two: QUIC uses a packet number for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with delivery order determined by stream offsets encoded within STREAM frames.

QUIC TLS

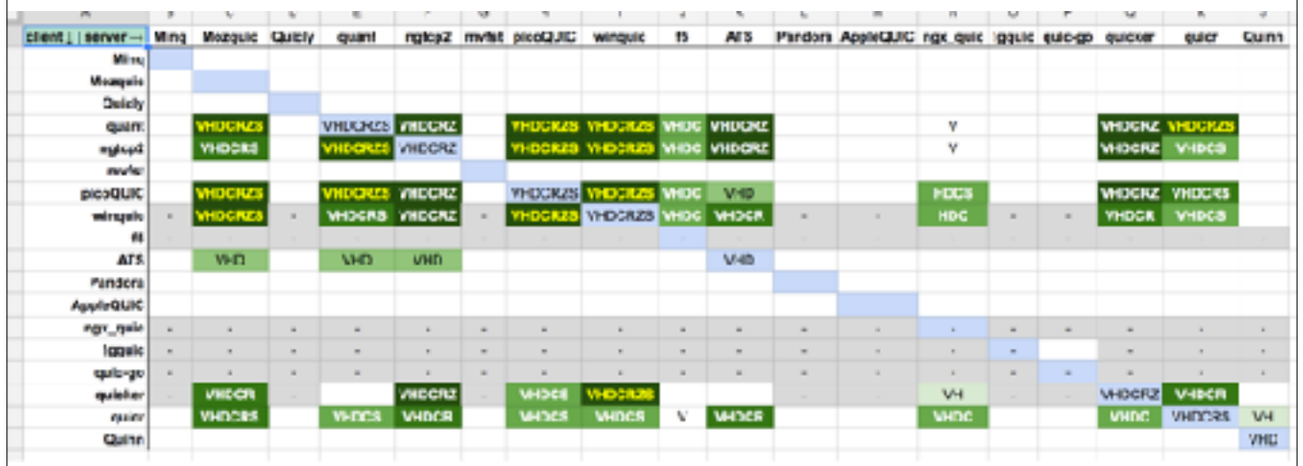
- TLS 1.3 handshake with a custom extension
  - Extension for announcing transport parameters
- After handshake, encrypt payloads with negotiated secret
  - Application data is not wrapped in TLS records
- Ongoing discussion about more TLS simplification

QUIC HTTP

- ALPN protocol "hq" negotiated during TLS handshake
- Builds on HTTP 2, differs in order to leverage QUIC
  - Different header compression scheme
  - Different framing mechanism

ALPN = Application-Layer Protocol Negotiation

Interoperability matrix



Version Negotiation — Handshake — Stream Data — Connection Close — Resumption — 0-RTT — Stateless Retry

Mozquic / mvfst / WinQuic / F5 / ATS / AppleQUIC / Cloudflare

Table of contents

- QUIC: problem statement, concepts and process
- **Rust ecosystem: rustls, *ring*, futures, tokio, h2, bytes**
- Quinn: design, implementation status, future

rustls

- Easy to work with/hack on
- QUIC transport parameters support:
  - Extension traits to add methods to Session types
  - Add `new_quic()` constructors to pass in local params
  - Add `get_quic_transport_parameters()` to fetch peer's
- Collateral contributions:
  - Added optional support for SSLKEYLOGFILE

ring, webpki

- Secure crypto, based on Google's BoringSSL
- rustls is built on this stack, so it's a good fit
  - Easy access from rustls cipher suite to *ring* algorithms

QUIC needs direct access to algorithms negotiated during the TLS handshake, so working with rustls and *ring* together makes that straightforward.

futures, tokio, h2

- Worked with basic futures in tokio-imap
- Looked for things to steal from the experts
  - How to create a good futures-based API?
  - How does the API layer interact with library internals?

bytes

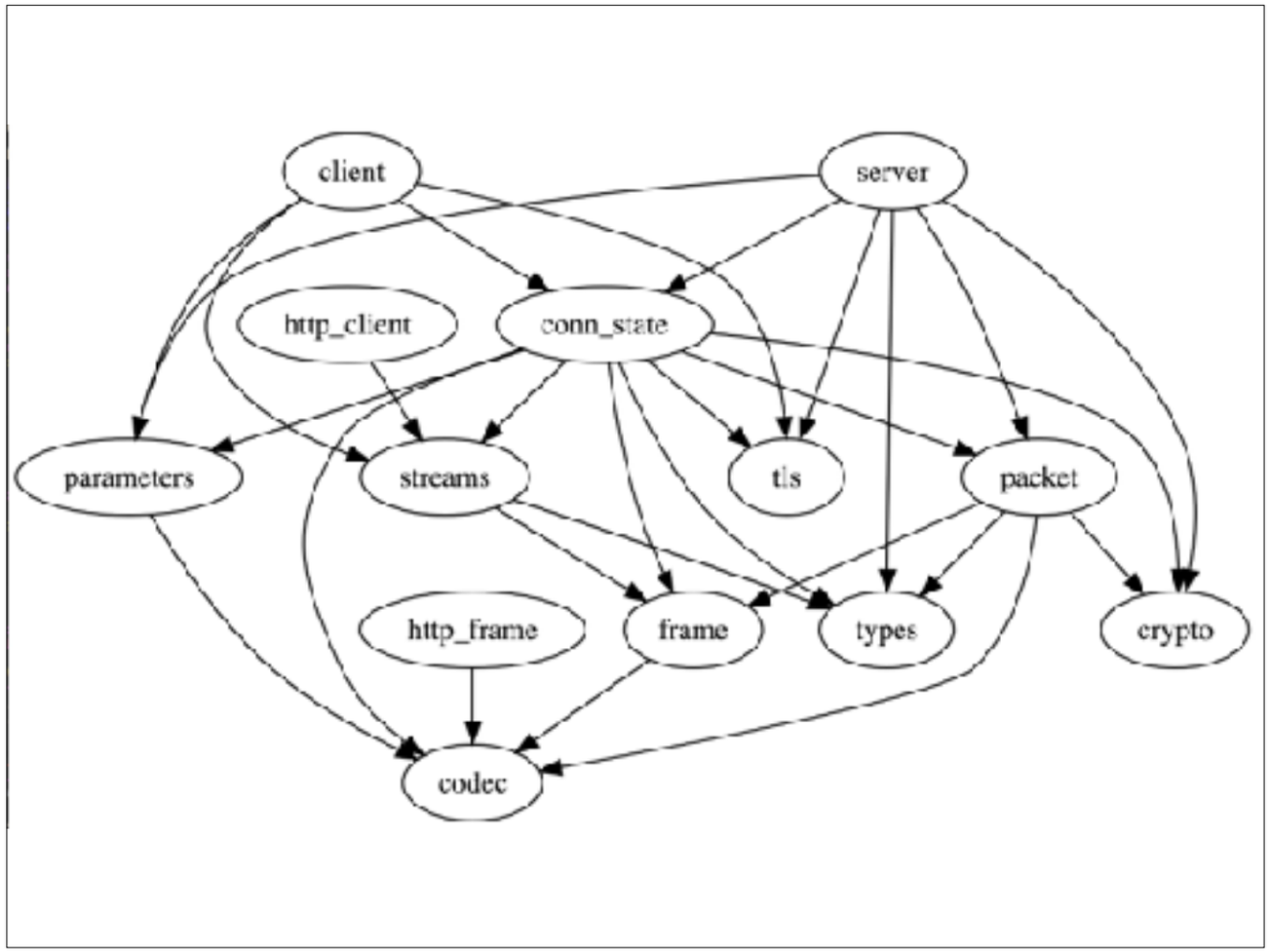
- Used `Buf` and `BufMut` traits to implement a `Codec` trait
- Based on the similar `Codec` trait seen in rustls
- Does all the low-level encoding & decoding

Table of contents

- QUIC: problem statement, concepts and process
- Rust ecosystem: rustls, *ring*, futures, tokio, h2, bytes
- **Quinn: design, implementation status, future**

Goals

- Implement the specifications faithfully
- Have clear abstractions & concepts
- Futures-based architecture — no polling
- Leverage the type system
- Good test coverage at different layers (currently ~85%)



Layered design

Implementation status

- Currently implements draft 11 w/ TLS 1.3 draft 28
- Client can handshake with other servers
- Other clients don't handshake with Quinn server so well
- Most implementations don't have much QUIC-HTTP yet
- Hope to keep this implementation moving forward

Draft 12 landed on Wednesday, but I haven't really looked into it. For the next draft, there will also likely be invasive changes to TLS handling, so Joe, hope you're happy to do more reviews.

Big job

- Packetization
- Flow control
- Recovery and retransmission
- Connection migration
- Error handling

Packetization implies PLPMTUD (PMTUD optional) [PMTU = Path Maximum Transmission Unit; PLPMTUD = Packetization Layer Path Maximum Transmission Unit Discovery]

quicr

- https://github.com/Ralith/quicr (by Benjamin Saunders)
- Discovered after I started Quinn
- Uses OpenSSL instead of rustls
- Will consider sharing code

Request for contributions

- https://github.com/djc/quinn
- Your contribution is much appreciated
- There are a few issues in the repository
- Happy to mentor anyone to help get started
- See you at the impl days?
- Or: contribute funding to help move the project forward