Fast QUIC sockets with vector packet processing

Aloys Augustin, Nathan Skrzypczak, Mathias Raoul, Dave Wallace
What is QUIC?
CALL IT TCP/2

ONE MORE TIME
The stack

HTTP/2
TLS
TCP
IP

HTTP/3
QUIC
UDP
Nice properties

- Encryption by default ~ TLS 1.3 handshake
- No ossification
- Built-in multiplexing
  - Very common application requirement
  - Independent streams in each connection
  - Addresses head-of-line blocking
  - Stream prioritization support
- Supports mobility
  - 5-tuple may change without breaking the connection
Conns & streams

Server

Stream #1
Stream #2

Client

Connection #1
Connection #2
Why QUIC - pros & cons

Pros

● Runs on UDP, can be implemented out of the kernel
● Addresses head of line blocking
● 5-tuple mobility
● Encryption by default

Cons

● Implementation complexity
● No standard northbound API (for now)
Still evolving relatively fast, not an IETF standard yet
A quick dive in the code
Building blocks

Socket API  L4 / UDP  QUIC implem

Cient app  Wire
Building blocks

- vpp client lib
- vpp
- vectorization
- fast L2-3-4
- pluggable sessions
- few assumptions
- very modular
- https://github.com/h2o/quicly
What is VPP?

- Fast userspace networking dataplane - https://fd.io/
- Open-source: https://gerrit.fd.io/r/q/project:vpp
- Extensible through plugins
- Multi-architecture (x86, ARM, ...), runs in baremetal / VM / container
- Highly optimized for performance (vector instructions, cache efficiency, DPDK, native crypto, native drivers)
- Feature-rich L2-3-4 networking (switching, routing, IPsec, ...)
- Includes a host stack with L4 (TCP, UDP) protocols

→ Great platform for a fast userspace QUIC stack
VPP Host stack (1/2)

- Generic session layer exposing L4 protocols
  - Socket-like APIs
- Fifos used to pass data between apps and protocols
- Internal API for VPP plugins
- Similar external API for independent processes available through a message queue
- Designed for high performance
  - Saturates 40G link with 1 TCP flow or 1 UDP flow
  - Performance scales linearly with number of threads
VPP Host stack (2/2)
QUIC App Requirements

Three types of objects:

- Listeners
- Connections
- Streams
Socket-like API for QUIC

Three types of sockets for listeners, connections and streams
Connection sockets are only used to connect and accept streams
Connection sockets cannot send or receive data
Building a QUIC stack in VPP

Message queue
Control events

Session
rx
tx

TCP / UDP

L2/L3

VCL

application

rx tx

vpp
Building a QUIC stack in VPP

Message queue
Control events

Session
rx
tx

VCL
rx
 tx

application

QUIC

Session
rx
tx

UDP
rx
tx

L2/L3

vpp
Zooming in

QUIC

Northbound interface in VPP session layer

callbacks

quicly

picotls

callbacks

Southbound interface to VPP session layer
Allows quicly to use VPP UDP stack
QUIC Consumption models in VPP

- The VPP QUIC stack offers 3 consumption models:
  - External apps: independent, use the external (socket) API
  - Internal apps: shared library loaded by VPP, use the internal API
  - Apps can use the quickly northbound API directly → As long as they use the VPP threading model

Northbound interface in the host stack is optional!
RX path

VPP UDP rx node

UDP session rx fifo

QUIC stream session rx fifo

Decrypted payload copy

Callback

Stream data available to quickly app

Stream data available to internal app

Stream data available to external app

App notification

MQ event

Session event

Buffer copy

Session event

quickly initial packet decoding

Connection matching

quickly packet decryption

App MQ

MQ event

App notification
Memory management and ACKs

- VPP fifos are fixed size. What if a sender sends more data than fifo size?
  - Before a packet is decrypted, we have no way to know which stream(s) it contains data for
    → We cannot check the available space in the receiving fifo
  - Once a packet is decrypted, Quicly does not allow us to drop it otherwise it will never be retransmitted
  - Fortunately, QUIC has a connection parameter called `max_stream_data`, which limits the in-flight (un-acked) data per stream sent by peer.
  - Setting this parameter to the fifo size solves this problem, as long as we ACK data only when it is removed from the fifo

- QUIC has several other connection-level settings to control memory usage:
  - Max number of streams
  - Total un-acked data for the connection
TX path

Stream data pushed by internal app

Session event

MQ event

App

MQ

Stream data pushed by external app

Payload copy

QUIC stream session tx fifo

Stream data pushed by quicly app

Session event

quickly packet generation

quickly packet encryption

Session event

UDP payload copy

UDP session tx fifo

Session event

VPP session node
Backpressure

- **UDP backpressure**: we limit the amount of packets generated by quicly so as not to overflow the UDP tx fifo

- **How does an app know it should wait before sending more data?**
  - When Quicly cannot send data as fast as the app provides it, it stops reading from the QUIC streams tx fifos
  - The app needs to check the amount of space available in the fifo before sending data
  - The app can subscribe to notifications when data is dequeued from its fifos
Threading model

- VPP runs either with one thread, or one main thread + n worker threads
- UDP packets assignment to threads is dependent on RSS
  - The receiving thread is unknown when the first packet is sent
  - UDP connections start on one thread and migrate when the first reply is received
  - The VPP host stack sends notifications when this happens
- QUIC sessions are opened only when the handshake completes, and thus do not migrate (as long as there are no mobility events - not yet supported)
- All QUIC streams are placed in the thread where their connection exists
How quick is it?
Performance: evaluation

For now: no canonical QUIC perf assessment tool

Custom iperf-like client/server benchmark tool

- Opens N connections
- Then opens n streams in each connection
- Client sends d bytes of data per stream
- Server closes the streams, then the connection

Typical setup N=10 n=10 d=1GB
Performance: test setup

- Core pinning, VPP and test apps on same NUMA node
- 1500 bytes MTU
- 2x Intel Xeon Gold 6146 3.2GHz CPUs
Performance: initial results

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10</td>
<td>1 worker</td>
<td>3.5 Gbit/s</td>
<td></td>
</tr>
<tr>
<td>100x10</td>
<td>4 workers</td>
<td>13.7 Gbit/s</td>
<td></td>
</tr>
</tbody>
</table>

Simultaneous connections

- Scales up to 100k streams / core
- Handshake rate $\sim$1500 / s / core
Performance: optimisations

● **Crypto**
  ○ Quicly uses picotls by default for the TLS handshake and the packet encryption / decryption
  ○ Picotls has a pluggable encryption API, which uses openssl by default for encryption
  ○ Using the VPP native crypto API yielded better results
  ○ Further improvements were obtained by batching crypto operations, using the Quicly offload API:
    ■ N packets are received and decoded
    ■ These N packets are decrypted at the same time
    ■ The decrypted packets are passed to quicky for protocol processing
    ■ The same idea is applied in the TX path as well

● **Congestion control**
  ○ The default congestion control (Reno) of quicky doesn’t reach very high throughputs
  ○ Fortunately, it is pluggable as well :)
Performance: new results

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Workers</th>
<th>Speed (Gbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10 pre-optimization</td>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>10x10 w/ batching &amp; native crypto</td>
<td>1</td>
<td>4.5 (+28%)</td>
</tr>
</tbody>
</table>

For now, most of the CPU time is spent doing crypto. Intel Ice Lake CPUs will accelerate AES and may move the bottleneck more towards the protocol processing.
What’s next
Next steps

- Performance optimisation
- Mobility support
- Continuous benchmarking - soon on https://docs.fd.io/csit/master/trending/index.html

If you want to get involved:

https://gerrit.fd.io/r/q/project:vpp  - code in src/plugins/quic/

If you want to try it, check out the example code in src/plugins/hs_apps/
(host stack apps)
Use cases

- Scalable HTTP/3 servers
- Scalable gRPC over QUIC servers
- QUIC VPN
  - Better than SSL VPN: mobility support, using one stream per flow allows to get rid of head of line blocking
  - As easy to deploy as an SSL VPN: only a certificate is needed on the server, with an authentication mechanism for clients
- QUIC VPN with transparent proxying
  - Transparently terminating the TCP connections at the VPN gateway and sending only the TCP payloads in QUIC streams allows to get rid of the nested congestion control issues
Takeaways

● Great experience with quicly
● VPP now provides an easy API to use QUIC
● Host stack proved to be extensible for new protocols
● VPP framework gave good performance boost to QUIC
  ○ Native crypto + vector processing
  ○ Still some effort required to reach max levels of performance
Thanks for listening

Any questions?
Backup slides
Building a QUIC stack - a QUIC dive

- connect
- close
- read

- accept
- data

- in/out of order packets
- close

mq

rx tx

rx tx

rx tx

quicly

picotls

rx tx

rx tx

rx tx