Moving fast at scale
Experience deploying IETF QUIC at Facebook

Subodh Iyengar
Luca Niccolini
Overview

- FB Infra and QUIC deployment
- Infrastructure parity between TCP and QUIC
- Results
- Future and current work
Anatomy of our load balancer infra

- **Edge Proxygen**: Edge POP closer to user
- **Origin Proxygen**: Datacenter closer to service
- **Internet**: HTTP 1.1 over QUIC
- **HHVM**: HTTP 2 over TCP
- **Backbone network**: HTTP 1.1 / HTTP3 over QUIC
Infra parity between QUIC and TCP

- QUIC requires unique infrastructure changes
- Zero downtime restarts
- Packet routing
- Connection Pooling
- Instrumentation
Zero downtime restarts

- We restart proxygen all the time
- Canaries, Binary updates
- Cannot shutdown all requests during restart
- Solution: Keep both old and new versions around for some time
Zero downtime restarts in TCP

Old proxygen

Accepted socket
Client 1

Accepted socket
Client 2

Accepted socket
Client 3

Listening socket
Zero downtime restarts in TCP

Old proxygen

Accepted socket
Client 1

Accepted socket
Client 2

Accepted socket
Client 3

Unix domain socket with SCM_RIGHTS and CMSG

Listening socket

New proxygen
Zero downtime restarts in TCP

Old proxygen

Accepted socket
Client 1

Accepted socket
Client 2

Accepted socket
Client 3

New proxygen

Listening socket

Accepted socket
Client 4

Accepted socket
Client 5
Zero downtime restarts in QUIC

Problems

- No listening sockets in UDP
- Why not SO_REUSEPORT
- SO_REUSEPORT and REUSEPORT_EBPF does not work on its own
Zero downtime restarts in QUIC

Solution

- Forward packets from new server to old server based on a "ProcessID"
- Each process gets its own ID: 0 or 1
- New connections encode ProcessID in server chosen ConnectionID
- Packets DSR to client
Zero downtime restarts in QUIC

Solution
Zero downtime restarts in QUIC

Solution

```
Old proxygen

GetProcessID

UDP socket 1

UDP socket 2

UDP socket 3

SO_REUSEPORT group

New proxygen

Choose PID = 1

PID = 0

Choose PID = 1

QUIC connection 1

QUIC connection 2
```
Zero downtime restarts in QUIC

Solution

Old proxygen

Takeover sockets

New proxygen

Unix domain socket with SCM_RIGHTS and CMSG

QUIC connection 1

QUIC connection 2

UDP socket 1

UDP socket 2

UDP socket 3
Zero downtime restarts in QUIC

Solution

- **Old proxygen**
  - QUIC connection 1
  - QUIC connection 2
  - Takeover sockets

- **New proxygen**
  - UDP socket 1
  - UDP socket 2
  - UDP socket 3

Enapsulated with original source IP

UDP packet
Zero downtime restarts in QUIC

Solution

Old proxygen

QUIC connection 1

QUIC connection 2

New proxygen

UDP packet

Mobile device
Results

packets forwarded during restart

packets dropped during restart
The Future

Coming to a 4.19 kernel near you

Introduce BPF_MAP_TYPE_REUSEPORT_SOCKETARRAY and BPF_PROG_TYPE_SK_REUSEPORT

From: Martin KaFai Lau <kafai-AT-fb.com>
To: <netdev-AT-vger.kernel.org>
Subject: [PATCH bpf-next 0/9] Introduce BPF_MAP_TYPE_REUSEPORT_SOCKETARRAY and BPF_PROG_TYPE_SK_REUSEPORT
Date: Wed, 8 Aug 2018 00:59:17 -0700
Message-ID: <20180808075917.3009181-1-kafai@fb.com>
Cc: Alexei Starovoitov <ast-AT-fb.com>, Daniel Borkmann <daniel-AT-iogearbox.net>, <kernel-team-AT-fb.com>
Archive-link: Article

https://lwn.net/Articles/762101/
Stable routing
Stable routing of QUIC packets

- We were seeing a large % of timeouts
- We first suspected dead connections
- Implemented resets, even more reset errors
- Could not ship resets
- We suspected misrouting, hard to prove
- Gave every host its unique id
- Packet lands on wrong server, log server id
- Isolate it to cluster level. Cause was misconfigured timeout in L3
Stable routing of QUIC packets

- We have our own L3 load balancer, katran. Open source
- Implemented support for looking at serverid
- Stateless routing
- Misrouting went down to 0
- We’re planning to use this for future features like multi-path and anycast QUIC
Stable routing of QUIC packets

- Now we could implement resets
- -15% drop in request latency without any change in errors
Connection pooling
Pooling connections

- Not all networks allow UDP
- Out of a sample size of 25k carriers about 4k had no QUIC usage
- Need to race QUIC vs TCP
- We evolved our racing algorithm
- Racing is non-trivial
Naive algorithm

- Start TCP / TLS 1.3 0-RTT and QUIC at same time
- TCP success, cancel QUIC
- QUIC success, cancel TCP
- Both error, connection error
- Only 70% usage rate
- Probabilistic loss, TCP middleboxes, also errors: ENETUNREACH
Let’s give QUIC a head start

- Let’s add a delay to starting TCP
- Didn’t improve QUIC use rate
- Suspect radio wakeup delay and middleboxes
- Still seeing random losses even in working UDP networks
What if we don’t cancel?

- Don’t cancel QUIC when TCP success
- Remove delay on QUIC error and add delay back on success
- Pool both connections, new requests go over QUIC
- Complicated, needed major changes to pool
- Use rate improved to 93%
- Losses still random, but now can use QUIC even if it loses
What about zero rtt?

- No chance to test the network before sending 0-RTT data
- Conservative: If TCP + TLS 1.3 0-RTT succeeds, cancel requests over QUIC
- Replay requests over TCP
What about happy eyeballs?

- Need to race TCPv6, TCPv4, QUICv6 and QUICv4
- Built native support for Happy eyeballs in mvfst
- Treat Happy eyeballs as a loss recovery timer
- If 150ms fires, re-transmit CHLO on both v6 and v4.
- v6 use rate same between TCP and QUIC
We have good tools for TCP
Where are the tools for QUIC?
Solution: We built QUIC trace
Schema-less logging: very easy to add new logs
Data from both HTTP as well as QUIC
All data is stored in scuba

Debugging QUIC in production
Debugging QUIC in production

- Find bad requests in the requests table from proxygen
- Join it with the QUIC_TRACE table
- Can answer interesting questions like
  - What transport events happened around the stream id
  - Were we cwnd blocked
  - How long did a loss recovery take
ACK threshold recovery is not enough
HTTP connections idle for most of time
In a reverse proxy requests / responses staggered ~TLP timer
To get enough packets to trigger Fast retransmit can take > 4 RTT

Results deploying QUIC

- Integrated mvfst in mobile and proxygen
- HTTP1.1 over QUIC draft 9 with 1-RTT
- Cubic congestion controller
- API style requests and responses
  - Requests about 247 bytes -> 13 KB
  - Responses about 64 bytes -> 500 KB
- A/B test against TLS 1.3 with 0-RTT
  - 99% 0-RTT attempted
# Results deploying QUIC

<table>
<thead>
<tr>
<th>Latency</th>
<th>p75</th>
<th>p90</th>
<th>p99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall latency</td>
<td>-6%</td>
<td>-10%</td>
<td>-23%</td>
</tr>
<tr>
<td>Overall latency for responses &lt; 4k</td>
<td>-6%</td>
<td>-12%</td>
<td>-22%</td>
</tr>
<tr>
<td>Overall latency for reused conn</td>
<td>-3%</td>
<td>-8%</td>
<td>-21%</td>
</tr>
</tbody>
</table>

Latency reduction at different percentiles for successful requests
What about bias?

Latency reduction at different percentiles for successful requests

<table>
<thead>
<tr>
<th>Latency</th>
<th>p75</th>
<th>p90</th>
<th>p99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency for later requests</td>
<td>-1%</td>
<td>-5%</td>
<td>-15%</td>
</tr>
<tr>
<td>Latency for rtt &lt; 500ms</td>
<td>-1%</td>
<td>-5%</td>
<td>-15%</td>
</tr>
</tbody>
</table>
Takeaways

- Initial 1-RTT QUIC results are very encouraging
- Lots of future experimentation needed
- Some major changes in infrastructure required
Questions?