Computer Networks: QUIC

Ankit Singla
Where we are in the course ...

Part 1: Overview & Principles

Part 2: Applications

Part 3: Transport

Part 4: Algorithms
Computer networks

Part 3: Transport

#1 What do we need in the transport layer?
#2 How do we build reliable transport?
#3 How does the Internet’s transport work?
#4 Sockets: the application ↔ transport interface
A new Internet protocol is being deployed!

QUIC

Quick UDP Internet Connections
A new Internet protocol is being deployed!

[The QUIC Transport Protocol: Design and Internet-Scale Deployment, Langley et. al, SIGCOMM 2017]

Approaching 35% of Google’s “egress”, 7% of Internet
A new Internet protocol is being deployed!

• First major transport change in ~30 years
• You’re already using it
• Not just Google
• Standardization ongoing at IETF
• Will inform HTTP/3 design
We present QUIC, a new transport designed from the ground up to replace most of the traditional HTTPS stack: HTTP/2, TLS, and TCP (Figure 1). We developed QUIC as a user-space transport with deployment and continued evolution of transport mechanisms. QUIC is widely deployed: it currently accounts for over 30% of Google's total egress traffic in bytes and playsbacks by 18.0% for desktop users and 15.3% for mobile users as shown in Figure 2. QUIC improves transport performance for HTTPS traffic and to enable rapid deployment and continued evolution of transport mechanisms. QUIC has been globally deployed at Google on thousands of servers and is used to serve traffic to a range of clients including a widely-used mobile video streaming YouTube app. We find that on average, QUIC reduces latency of Google Search responses by 8.0% for desktop users and decreases latency of Google Search app on Android. We launched an early version of QUIC as an experiment in 2013. After several iterations with the protocol and following our deployment experience over three years, an IETF working group was formed to standardize it [2]. QUIC is a single monolithic protocol in employment and continued evolution of transport mechanisms. QUIC eliminates head-of-line blocking delays by using a lightweight data-structuring abstraction, IP addresses, which are multiplexed within a single connection so that changes to occur at application update timescales. The use of UDP as a substrate. Building QUIC in user-space facilitated its deployment as part of various applications and enabled iterative experiments on QUIC, performance improvements, and low-latency transport protocol designed from the ground up to be a single monolithic protocol in the traditional HTTPS stack.

Strategy: rebuild everything on UDP

[The QUIC Transport Protocol: Design and Internet-Scale Deployment, Langley et. al, SIGCOMM 2017]
Strategy: rebuild everything on UDP

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sequence number
- Acknowledgment
- HdrLen
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)

![UDP Header Diagram]

- SRC port
- DST port
- checksum
- length

DATA
Why build a new transport?

• Application context has changed
• TCP’s “hellos” are too expensive
• TCP doesn’t handle mobility
• Difficult to evolve TCP
  • Middleboxes, firewalls are suspicious, fickle
Why build a new transport?

- Application context has changed
- TCP’s “hellos” are too expensive
- TCP doesn’t handle mobility
- Difficult to evolve TCP
  - Middleboxes, firewalls are suspicious, fickle
Redesigning transport for the Web

HTTP/1.1

TCP 1
TCP 2
TCP 3

HTTP/2

Stream 1
Stream 2
Stream 3

Innovating Transport with QUIC: Design Approaches and Research Challenges
Cui et al., IEEE Internet Computing 2017.
Sliding window halts if any flow halts

Sending process

Last byte written

Receiver Window

Receiver Window

TCP
Redesigning transport for the Web

HTTP/1.1

TCP 1
TCP 2
TCP 3

HTTP/2

Stream 1
Stream 2
Stream 3

QUIC

Stream 1
Stream 2
Stream 3

Innovating Transport with QUIC: Design Approaches and Research Challenges
Cui et al., IEEE Internet Computing 2017.
Why build a new transport?

- Application context has changed
- TCP’s “hellos” are too expensive
- TCP doesn’t handle mobility
- Difficult to evolve TCP
  - Middleboxes, firewalls are suspicious, fickle
Eliminating the “hello” safely (in the common case)

DNS  “Hi”  Data transfer

- Establish connection
- TCP Syn
- TCP syn + ack
- TCP ack + HTTP GET
- Request response
- Close connection
- Data transfer

handshake state in a “transport cookie”
QUIC rolls in TCP and TLS connection negotiation

First connection to a server

Innovating Transport with QUIC: Design Approaches and Research Challenges
Cui et al., IEEE Internet Computing 2017.
QUIC rolls in TCP and TLS connection negotiation

Repeat connection

Innovating Transport with QUIC: Design Approaches and Research Challenges
Cui et al., IEEE Internet Computing 2017.
But 0-RTT data transfer is not entirely “free”

Open to replay attacks

| 0-RTT key-exchange messages | 0-RTT data "request" | process "request"
|-----------------------------|----------------------|------------------|
| accept 0-RTT key-exchange response messages | enforce loss of state (e.g., reboot) | reject after state loss for security reasons
| replay 0-RTT key-exchange messages | replay 0-RTT data "request" | reject 0-RTT key-exchange response messages
| final key exchange messages | resend data "request" under final key (to ensure reliable transmission) | process "request"

Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates
This problem is fundamental

- Standard method of avoiding replay is “nonces”
- Generate a random number (nonce)
- Ask the other party to encode it with key
- Check the encoded number = your nonce
- Proof that they responded to this message

If the server could send a challenge nonce, problem would be solved
How do we solve this problem?

Maintain global and temporal consistency on server side
Works by detecting replays (but hard to implement)

Don’t solve the problem 1: only idempotent operations with 0-RTT
Use new key for subsequent, non-idempotent operations

Don’t solve the problem 2: don’t provide reliability with 0-RTT
But just throw the problem to application level. No replay, no problem.
Why build a new transport?

- Application context has changed
- TCP’s “hellos” are too expensive
- **TCP doesn’t handle mobility**
- Difficult to evolve TCP
  - Middleboxes, firewalls are suspicious, fickle
TCP doesn’t handle mobility

• TCP: Connection is specific to src.IP
• UDP: gimme dest.IP, port and I’ll handle it
• QUIC is built on UDP
• Adds “connection IDs” separate from IP addresses
  • Initialized (randomly) in handshake
Why build a new transport?

- Application context has changed
- TCP’s “hellos” are too expensive
- TCP doesn’t handle mobility
- Difficult to evolve TCP
  - Middleboxes, firewalls are suspicious, fickle
Middleboxes trying to “help”

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Sequence number |                  |
| Acknowledgment  |                  |

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packets using these are weird, let’s throw them!
Strategy: rebuild everything on UDP

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
</tbody>
</table>

DATA
Strategy: rebuild everything on UDP

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sequence number
- Acknowledgment
- HdrLen
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)

DATA

**QUIC public headers**

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- checksum
- length

Carry things like Connection ID, setup flags, QUIC version …
**Strategy: rebuild everything on UDP**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

DATA

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
</tbody>
</table>

QUIC public headers

**Middlebox vendor:** “if these bits have these values, it’s QUIC, let it pass”