20 QUIC Dissection

Using Wireshark to Understand QUIC Quickly

Megumi Takeshita

ikeriri network service

supplemental files http://www.ikeriri.ne.jp/sharkfest

ParkSuite Classroom
11 November 2017
11:15am-12:30pm

#sf17eu • Estoril, Portugal • 7-10 November 2017
Megumi Takeshita, ikeriri network service

- Founder, ikeriri network service co., ltd
- Wrote 10+ books about Wireshark
- Reseller of Riverbed Technology (former CACE technologies) in Japan
- Attending all Sharkfest
- Translator of QT Wireshark into Japanese
In this presentation, Megumi explains the details of QUIC, and shows you how to understand the protocol and mechanisms involved. Using sample trace files, Megumi will show how to inspect and visualize QUIC traffic and explain the advantage of QUIC in comparison with other protocols too.

NOTE: IQUIC (IETF QUIC) is Internet-Draft and now standardizing, so some specification may be changed and the sample trace file is not adequate.
Set up environment

• For QUIC dissection, we need nightly build version of Wireshark (this time I use 2.5.0-1547-gbe625b9b development version)
• All supplemental files of this presentation is below http://www.ikeriri.ne.jp/sharkfest (temporal)
Open simple HTTP/1.1

- open httpikeriri.pcapng of simple HTTP/1.1 packet,
- HTTP/1.1 request response loop
- Head of Line blocking
- Rich application needs many TCP connection (AJAX)
HTTP/1.1 is difficult to speed up

- HTTP request have to send after previous response has been received.
- Please input display filter in Wireshark “http.next_request_in” (Next request in frame in HTTP request)
- HTTP request is always waiting in one connection. (head line blocking)
- Display filter “http” and Statistics>Flow Graph
HTTP/1.1 is text based, not efficient protocol

- Right click HTTP header and “follow http stream”
- HTTP is text-based application protocol, easy to read, but not efficient, ambiguous, and redundant
- HTTP messages are clear texts so they uses more data and CPU power for dissection.
- Many connections are separated by each other TCP connection, they work their own TCP rules without HTTP.
Wider bandwidth, Faster computing in today's internet, then what is the protocol?
HTTP/1.0 (RFC1945-1996)
HTTP/1.1 (RFC2068-1997)
New generation of web protocol comes
HTTP/2.0 (RFC7540-2015) former SPDY
Google, Facebook, Twitter, Yahoo, and major website using Chrome, Edge, Safari and major browser
Set SSLKEYLOGFILE variable to decrypt SSL/TLS

Open Chrome URL “chrome://flags/” and disable QUIC protocol in list box, now Chrome prefer to use HTTP2

Start capture and open www.twitter.com, type chrome://net-internals/#http2 you can see the HTTP/2 sessions

This time open twitter.pcapng and set (Pre)-Master-Secret log filename Twitter_unencrypted_premaster_secret.txt in SSL preference
HTTP/2.0 uses binary frame with Huffman coding compression in a SSL/TLS connection

- Set “http2.header” in display filter and check the #14
- The packet contains EthernetII, IPv4, TCP, SSL, and HTTP2 header
  - Application: HTTP/1.1 semantics
  - Session: HTTP/2.0
  - Session: SSL/TLS
  - Transport: TCP
- HTTP/2.0 uses binary frame with Huffman coding, check packet bytes pane
Connection process of HTTP/2.0

- Click Statistics > Flow Graph and check connection process of HTTP/2.0
- HTTP/2.0 needs TCP 3 way handshake that contains 1 RTT (round trip time) SYN-SYN/ACK-ACK from Client side
- HTTP/2.0 needs SSL/TLS connection that contains 2 RTT (round trip time) from Client side
  Client Hello/Server Hello-Certificate-Server Key Exchange-Server Hello Done/Client Key Exchange-New Session Ticket (TLS)-Change Cipher Spec-Finished at the first time
- We need TCP 1 and SSL/TLS 2 RTT at the first time
HTTP/2.0 Stream mechanism

HTTP/2.0 uses 1 tcp connection and many Stream (virtual connection channel) that has id and priority.
HTTP/1.1

TCP connection

SYN

1 RTT

SYN/ACK

ACK

GET / HTTP/1.1

HTTP/2.0

TCP connection

SYN

1 RTT

SYN/ACK

ACK

GET / HTTP/2.0

SSL/TLS1.0 connection

Client Hello

1 RTT

Server Hello-Certificate-Server Key Exchange-Server Hello Done

1 RTT

Client Key Exchange-Change Cipher Spec-Finished

New Session Ticket

Change Cipher Spec-Finished
GQUIC

• Google creates proprietary protocol, QUIC (Quick UDP Internet Connection) (a.k.a. GQUIC)

• GQUIC omits TCP, SSL/TLS and HTTP/2.0 and provides a monolithic mechanism of TCP + SSL/TLS authentication and encryption + HTTP/2 multiplexing and compression in UDP stream

• Already used in Google service (Gmail, Youtube, ...)

• QUIC needs just 1-RTT at the first time, and no RTT (0-RTT) when we connect again (if resumption successes)
• Open imfeelinglucky.pcapng, it is the packet that just I pushed I’m feeling lucky button at google using Chrome

• At this time we just see some UDP streams of QUIC

• Open the Chrome and type chrome://net-internals/#quic you can see current QUIC sessions

![Image of Wireshark interface showing QUIC streams]

<table>
<thead>
<tr>
<th>Host</th>
<th>Version</th>
<th>Peer address</th>
<th>Connection UID</th>
<th>Active stream count</th>
<th>Active streams</th>
<th>Total streams count</th>
<th>Packets Sent</th>
<th>Packets Lost</th>
<th>Packets Received</th>
<th>Connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>nyc.google.com:443</td>
<td>QUIC_VERSION_35</td>
<td>123.45.67.89:80</td>
<td>1234567890123456</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>client4.google.com:443</td>
<td>QUIC_VERSION_35</td>
<td>98.76.54.32:80</td>
<td>234567890123456</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>fonts.gstatic.com:443</td>
<td>QUIC_VERSION_35</td>
<td>123.45.67.89:80</td>
<td>1234567890123456</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>#1.googleusercontent.com:443</td>
<td>QUIC_VERSION_35</td>
<td>123.45.67.89:80</td>
<td>1234567890123456</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>True</td>
<td></td>
</tr>
</tbody>
</table>
Check GQUIC packets

- Check header encapsulation (Ethernet II, IP, UDP, and QUIC) and payloads are encrypted

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000000</td>
<td>10.0.0.13</td>
<td>172.217.25.180 QUIC</td>
<td>175 SH,</td>
<td>180</td>
<td>Protected Payload (KΦ0), PKN: 205</td>
</tr>
<tr>
<td>2</td>
<td>0.012084</td>
<td>172.217.25.180</td>
<td>10.0.0.13</td>
<td>QUIC</td>
<td>469 SH,</td>
<td>Protected Payload (KΦ0), PKN: 7</td>
</tr>
<tr>
<td>3</td>
<td>0.015205</td>
<td>172.217.25.180</td>
<td>10.0.0.13</td>
<td>QUIC</td>
<td>94 SH,</td>
<td>Protected Payload (KΦ0), PKN: 205</td>
</tr>
</tbody>
</table>

- Frame 1: 175 bytes on wire (1400 bits), 175 bytes captured (1400 bits) on interface 0
- Ethernet II, Src: 458291c5f45f (00:00:c0:0e:0a:5f), Dst: 808000:0f1b:06a:5f
- User Datagram Protocol, Src Port: 50278, Dst Port: 443
- QUIC (Quick UDP Internet Connections) IETF
  - 0... .... = Header Form: Short Header (0)
  - .0... .... = Connection ID Flag: False
  - ..0... .... = Key Phase Bit: False
  - ...01100 = Packet Type: Unknown (12)
  - Packet Number: 206
  - Protected Payload: d7f8e5231a78006c86a53a1f6e85a50765bf2cc2b8aca...

- This is not a first connection, so it immediately starts data transaction (0-RTT) because we can see SH(Short Header) at Header Form field.
- 64-bit packet number is used as a part of nonce. Each endpoint uses a separate packet number, that is increasing.
QUIC (IETF Quick UDP Internet Connection)

- Now IETF standardize IETF QUIC (a.k.a. IQUIC)
- IQUIC also provides a monolithic mechanism of TCP reliable transport + SSL/TLS1.3 authentication and encryption + HTTP/2 multiplexing and compression
- Data tracker (IETF) https://datatracker.ietf.org/wg/quic

TCP+SSL/TLS+HTTP/2.0=QUIC
IETF QUIC standards

- Working Group
  https://github.com/quicwg

- Internet-Draft (October, 2017)

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-07

J. Iyengar, Ed. Google
M. Thomson, Ed. Mozilla

October 13, 2017

Core specification
IETF QUIC standards

- QUIC-TLS (October, 2017)

Using TLS in QUIC
Open sample packets of IETF QUIC

- Open `quic_ietf_draft05_ngtcp2.pcapng` using Wireshark (Thank you Alexis-san for dissector and sample pcap file)

- View > Coloring rules..., new rule name: UDP source port 443, set filter `udp.srcport==443`, and set pink color at background

- Blue color is from Client and Pink is from Server
Long header of QUIC

• Click #1 packet and check QUIC header format

User Datagram Protocol, Src Port: 39916, Dst Port: 443
QUIC (Quick UDP Internet Connections) IETF
1... .... = Header Form: Long Header (1)
0.00 0010 = Packet Type: Client Initial (2)
Connection ID: 0x8ee4cfa7e9f5d9c
Packet Number: 558625387
Version: draft-05 (0xff00006:)
STREAM Stream ID: 0
  Frame Type: STREAM (0xc1)
    11... .... = Stream: 0x3
    ..0. .... = Fin(F): False
    ...0 0... = Stream Length (SS): 1 Byte (0)
    .... .00. = Offset Length (OO): 0 Byte (0)
    .... ...1 = Data Length (D): 2 Bytes
    Stream ID: 0 (Cryptographic handshake)
    Data Length: 274
    Stream Data: 160301010d0100001090303a9b79ac62f9f
> Secure Sockets Layer
> PADDING Length: 948
> Hash: 4e8c7f5146059fb9

64-bit random connection ID from the client

Long headers are used for negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender switches to send short header.

Packet type indicates the frame type of QUIC

64-bit packet number is used as part of nonce. Each endpoint uses a separate packet number, that is increasing.

Stream is the same mechanism of HTTP/2.0 stream, and Stream ID 0 is reserved for cryptographic handshake (TLS1.3).
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Version Negotiation</td>
<td>Server sends this type packet for not supporting client’s version (Long header)</td>
</tr>
<tr>
<td>0x02</td>
<td>Client Initial</td>
<td>Client sends this type packet for initializing handshake (Long Header)</td>
</tr>
<tr>
<td>0x03</td>
<td>Server Stateless Retry</td>
<td>Server sends this type packet as cryptographic handshake message and ACK for requiring a new Client Initial packet (Long Header)</td>
</tr>
<tr>
<td>0x04</td>
<td>Server Cleartext</td>
<td>Server sends this type packet as cryptographic handshake message and ACK that contains server chosen connection ID and randomized packet number with STREAM, PADDING, ACK. (Long Header)</td>
</tr>
<tr>
<td>0x05</td>
<td>Client Cleartext</td>
<td>Client sends this type packet as the receipt of Server Cleartext message, Client Cleartext contains Server selected connection ID and incremented packet number of Client Initial with STREAM, PADDING, ACK. (Long Header)</td>
</tr>
<tr>
<td>0x06</td>
<td>0-RTT Protected</td>
<td>Packets that are protected with 0-RTT keys are sent with Long Header; all packets protected with 1-RTT keys are sent with Short Header. Packets protected with 0–RTT keys use a type value of 0x06. The connection ID field for a 0–RTT packet is selected by the client.</td>
</tr>
</tbody>
</table>
• Click connection ID field, right click and “Apply as column” (same as packet number) in #1 packet, and check the changes of both

• Server set 64-bit the random connection ID in #2 packet, Client updates the connection ID as the same number

• Packet number is set randomly (0 and 2^{31}-1) and used as a part of nonce. Each endpoint uses a separate packet number, that is increasing

<table>
<thead>
<tr>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Connection ID</th>
<th>Packet Number</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>1294</td>
<td>0x8ee4cfafe7e9f5d9c</td>
<td>558625387 LH, Client Initial, PKN: 558625387, CID: 0x8ee4cfafe7e9f5d9c</td>
<td></td>
</tr>
<tr>
<td>0.03...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>1283</td>
<td>0x5ab560082f4e162c</td>
<td>726976297 LH, Server Cleartext, PKN: 726976297, CID: 0x5ab560082f4e162c</td>
<td></td>
</tr>
<tr>
<td>0.03...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>221</td>
<td>0x5ab560082f4e162c</td>
<td>726976298 LH, Server Cleartext, PKN: 726976298, CID: 0x5ab560082f4e162c</td>
<td></td>
</tr>
<tr>
<td>0.03...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>159</td>
<td>0x5ab560082f4e162c</td>
<td>558625388 LH, Client Cleartext, PKN: 558625388, CID: 0x5ab560082f4e162c</td>
<td></td>
</tr>
<tr>
<td>0.03...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>83</td>
<td>0x5ab560082f4e162c</td>
<td>726976299 SH, Protected Payload (KPO), PKN: 726976299, CID: 653624811709570012</td>
<td></td>
</tr>
<tr>
<td>4.45...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>91</td>
<td>0x5ab560082f4e162c</td>
<td>558625389 SH, Protected Payload (KPO), PKN: 558625389, CID: 653624811709570012</td>
<td></td>
</tr>
<tr>
<td>4.45...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>106</td>
<td>0x5ab560082f4e162c</td>
<td>726976300 SH, Protected Payload (KPO), PKN: 726976300, CID: 653624811709570012</td>
<td></td>
</tr>
<tr>
<td>4.45...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>83</td>
<td>0x5ab560082f4e162c</td>
<td>726976301 SH, Protected Payload (KPO), PKN: 726976301, CID: 653624811709570012</td>
<td></td>
</tr>
<tr>
<td>4.45...</td>
<td>127...</td>
<td>127...</td>
<td>QUIC</td>
<td>83</td>
<td>0x5ab560082f4e162c</td>
<td>558625390 SH, Protected Payload (KPO), PKN: 558625390, CID: 653624811709570012</td>
<td></td>
</tr>
</tbody>
</table>
Stream ID (encrypted in Short Header)

- QUIC packet has a 32-bit STREAM id for multiplexing many data connections.
- Clients use odd-number, Server use even-number, 0 is reserved for cryptographic Handshake (usually TLS connection).
- QUIC stream mechanism is almost the same as HTTP/2.0(also as TCP).
- Stream change the state, Many streams in a UDP connection.
Short header of QUIC

- Click #5 packet and check QUIC IETF header

```
Frame 5: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on interface 0
Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 443, Dst Port: 39916
QUIC (Quick UDP Internet Connections) IETF
0... ..... = Header Form: Short Header (0)
.1... ..... = Connection ID Flag: True
..0. ..... = Key Phase Bit: False
...0 0011 = Packet Type: 4 octet (3)
Connection ID: 0x5ab56b082f4e162c
Packet Number: 726976299
Protected Payload: a48cd45b995a53917485448d31ac728354b314e876ed4e23
```

- Set 0: Connection ID field is omitted
- Set 1: Connection ID field is present

- 64-bit random Server chosen connection ID
- Every time that a new set of keys is used for protecting outbound packets, the KEY_PHASE bit in the public flags is toggled.
- 64-bit packet number is used as part of nonce. Each endpoint uses a separate packet number, that is increasing.

The short header can be used after the version and 1-RTT keys are negotiated.
How to negotiate and install session key in IQUIC

- IQUIC is learned from SSL/TLS to install session key, but how do QUIC install session key at the first time (1-RTT) and at resumption (0-RTT)?
- Open tls10ikeriri.pcapng to remember how to negotiate and install session key in TLS1.0
- tls10ikeriri.txt is a PEM format certification file with server’s private key
- Set RSA key list in SSL preference of Wireshark
Open `tls10ikeriri.pcapng` and set RSA key list (`tls10ikeriri.txt`).
Quick Dissection Using Wireshark to Understand QUIC Quickly

Key creation process of TLS1.0

1. **Client Hello**

2. **Server Hello-Certificate-Server Key Exchange-Server Hello Done**

3. **Client Key Exchange-Change Cipher Spec-Finished**

4. **Encrypt Premaster secret**

   - **Client Nonce (Random)**
   - **Server cert**
   - **Encrypted pre-master secret**
   - **Master secret**
   - **MAC secret**
   - **Session Key**

5. **Decrypt encrypted Premaster secret**

   - **Server cert**
   - **Server Public Key**
   - **Server Private Key**
   - **Encrypted pre-master secret**
   - **Unencrypted pre-master secret**
   - **Master secret**
   - **MAC secret**
   - **Session Key**
Filter “ssl” and check the each TLS packet

- Check packet #6 and expand Client Key Exchange

Client send
Encrypted
PreMaster
Secret after
negotiated
with Server
TLS1.0/1.2 needs 2 RTT at the first connection

TCP connection

SYN
1 RTT
ACK
SYN/ACK

SSL/TLS1.0 connection

Client Hello
1 RTT
Server Hello-Certificate-Server Key Exchange-Server Hello Done
1 RTT
Client Key Exchange
-Change Cipher Spec-Finished
New Session Ticket
Change Cipher Spec-Finished

HTTP/2.0

- Old TLS needs 2 RTT at the fist connection
- It is not use for QUIC 1RTT connection
- Another way to negotiate and install session key....
TLS1.3 Internet Draft 21

• New TLS protocol since 2014 now Internet-Drafts

• Stronger (few cleartext) and Faster (few packet)

• New encryption / authentication

• No SessionID, No Ticket, use PSK
  No Change Cipher Spec,
  No Client Key Exchange,

• 1-RTT at first time, 0-RTT when we connect again
Sample trace of TLS1.3

- Open sample trace file sip.pcap from Wireshark Wiki sip-tls-1.3-and-rtcp.zip SIP call over TLS 1.3 transport with enabled RTCP. Used openssl 1.1.1 prerelease version (https://wiki.wireshark.org/SampleCaptures)

- Open sip.pcap and filter ssl in Display Filter

- Statistics > Flow Graph and set Displayed Packet to see the 1-RTT full handshake of TLS1.3
Open sip.pcapng and filter ssl and create Flow Graph
TLS1.3 1-RTT handshake

There are no Client Key Exchange, no Change Cipher Spec packet, and the encryption starts after Server Hello.

The other handshake is encrypted using PSK (Pre Shared Key).

Client send Application data after receiving Server packet.

It needs just 1 Round trip time from Client side.

TCP connection

SYN

1 RTT

SYN/ACK

ACK

TLS1.3 handshake

Client Hello

1 RTT

Server Hello, Application Data...

Finished, Application Data

(Application Data...)

The other handshake is encrypted
(Encrypted Extension / Server Configuration / Certificate / Certificate Verify / Finished)
Client Hello
(matches Client Key Exchange, Change Cipher Spec)

Extension: `psk_key_exchange_modes` (len=2)
  Type: `psk_key_exchange_modes` (45) Length: 2
  PSK Key Exchange Modes Length: 1
  PSK Key Exchange Mode: PSK with (EC)DHE key establishment (`psk_dhe_ke`) (1)

Extension: `key_share` (len=71)
  Type: `key_share` (40) Length: 71
  Key Share extension
  Key Share Key Length: 69
  Key Share Entry: Group: `secp256r1`, Key Exchange length: 65
  Group: `secp256r1` (23)
  Key Exchange Length: 65
  Key Exchange: `04f145e0e15072f4983d04be08c7886c598af98607204dd0...`

Extension: `certificate Authorities` (len=40)
  Type: `certificate Authorities` (47) Length: 40
  Distinguished Names Length: 38
  Distinguished Names (38 bytes)
Server Hello

(matches former Change Cipher Spec)

Handshake Protocol: Server Hello

Handshake Type: Server Hello (2)
Length: 111
Version: TLS 1.3 (draft 21) (0x7f15)
Random: 8f3a63a080b3c1ae2b3192c75f12d4f28afdbf1f123a68f81...
Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
Extensions Length: 73
Extension: key_share (len=69)

Type: key_share (40)
Length: 69
Key Share extension

Key Share Entry: Group: secp256r1, Key Exchange length: 65
Group: secp256r1 (23)
Key Exchange Length: 65
Key Exchange: 04110e96ae58d23b968ebb7fd9075d83348733a622013785...

TLSv1.3 Record Layer: Application Data Protocol: sip.tcp

Set Key Share settings

Determined Auth/Encryption

Send Key Exchange Data

Send Server Nonce

The others are encrypted
TLS1.3 in IETF QUIC

• Let’s go back to quic_ietf_draft05_ngtcp2.pcapng

• Check #1 packet of Client Initial (including Client Hello)
  Extension: quic_transport_parameters
  Extension: psk_key_exchange_modes
  Extension: key_share

• Check #2 packet of Server Cleartext (including Server Hello)
  Extension: key_share

• #3 (Server Cleartext) and #4(Client Cleartext) is encrypted with application data (http-over-tls)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>Server Cleartext</td>
</tr>
<tr>
<td>0x05</td>
<td>Client Cleartext</td>
</tr>
</tbody>
</table>
Quick Dissection Using Wireshark to Understand QUIC Quickly

- **Client Hello/Server Hello of IQUIC**
- **Secure Sockets Layer**
- **TLSv1.3 Record Layer: Handshake Protocol: Client Hello**
  - Content Type: Handshake (22)
  - Version: TLS 1.3 (0x3F00)
  - Length: 265
  - Handshake Protocol: Client Hello
    - Handshake Type: Client Hello (1)
    - Version: TLS 1.2 (0x0203)
    - Random: 2af975ac6279d8786f6322468f93cd73d166c284526ff42d...
    - Session ID Length: 8
    - Cipher Suits Length: 8
    - Cipher Suites (6 suites): 1
    - Compression Methods Length: 1
    - Extensions Length: 216
    - Extension: tlsext_psk_client_identity (len=14)
    - Extension: tlsext_pre_shared_key (len=14)
    - Extension: tlsext_psk_ciphersuites (len=4)
    - Extension: tlsext_psk_groups (len=4)
    - Extension: tlsext_psk_algorithms (len=4)
    - Extension: tlsext_psk_key_usage (len=4)
    - Extension: tlsext_psk_key_share (len=4)
    - Extension: tlsext_psk_renegotiation_info (len=4)
    - Extension: tlsext_psk_server_name (len=4)
    - Extension: tlsext_psk_extensions (len=4)
    - Extension: tlsext_server_identity (len=14)
    - Extension: tlsext_psk_ciphersuites (len=4)
    - Extension: tlsext_psk_groups (len=4)
    - Extension: tlsext_psk_algorithms (len=4)
    - Extension: tlsext_psk_key_usage (len=4)
    - Extension: tlsext_psk_key_share (len=4)
    - Extension: tlsext_psk_renegotiation_info (len=4)
    - Extension: tlsext_psk_server_name (len=4)
    - Extension: tlsext_psk_extensions (len=4)
    - Extension: tlsext_server_identity (len=14)
  - Key Share extension
    - Key Share Entry: Group: secp256r1, Key Exchange length: 65
  - Send Key Exchange Data
  - The others are encrypted
  - Send Client Nonce
  - Send Server Nonce
  - Set PSK Key Exchange Mode
  - Set Key Share settings
Quick Dissection Using Wireshark to Understand QUIC Quickly

- **Check #5- packets with Short Header of IQUIC**
- **The short header can be used after the version and 1-RTT keys are negotiated.**

Transactions are independent and based on IP/UDP

Next time Client try to use 0-RTT way.
Comparision between HTTP/1.1, HTTP/2.0 and IETF QUIC

HTTP/1.1
- SYN
- 1 RTT
- SYN/ACK
- ACK
- GET / HTTP/1.1
- X many times

HTTP/2.0 with SSL/TLS
- SYN
- 1 RTT
- SYN/ACK
- ACK
- Client Hello
- 1 RTT
- Server Hello-Certificate-Server Key Exchange-Server Hello Done
- Client Key Exchange
- Change Cipher Spec-Finished
- New Session Ticket
- Change Cipher Spec-Finished
- 1 RTT

IETF QUIC handshake
- Client Hello
- 1 RTT
- Server Hello, Application Data...
- Finished, Application Data
- (Application Data...)
- The other handshake is encrypted
  (Encrypted Extension / Server Configuration / Certificate / Certificate Verify / Finished)

Quick Dissection Using Wireshark to Understand QUIC Quickly
Use Wireshark!

Thank you very much!!

どうもありがとうございます！