Une courte session QUIC avec explications

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 28 mai 2021

https://www.bortzmeyer.org/quic-demo.html

On va voir ici quelques exemples QUIC <https://www.bortzmeyer.org/quic.html>, avec explications des paquets échangés. On saute directement dans les détails donc, si ce n'est pas déjà fait, je recommande de commencer par l'article d'introduction à QUIC <https://www.bortzmeyer.org/quic.html>.

Comme on ne peut pas facilement utiliser tcpdump ou Wireshark (mais lisez cet article jusqu'au bout, j'explique à la fin) pour analyser une communication QUIC <https://www.bortzmeyer.org/quic.html> (tout est systématiquement chiffré), on va se servir d'un client QUIC qui a la gentillesse d'afficher en détail ce qu'il fait, en l'occurrence picoquic <https://github.com/private-octopus/picoquic>. Pour le compiler :

```
# La bibliothèque picotls
git clone https://github.com/h2o/picotls.git
cd picotls
git submodule init
git submodule update
cmake .
make
cd ..

# picoquic
git clone https://github.com/private-octopus/picoquic.git
cd picoquic
cmake .
make
```

Puis on va utiliser le programme client `picoquicdemo` pour se connecter en HTTP/3 à un serveur public, mettons `f5quic.com` :

```
% ./picoquicdemo -l /tmp/quic.txt f5quic.com 4433 '0:index.html'
```

La ligne de commande ci-dessus lance `picoquicdemo`, enregistre les informations dans le fichier `/tmp/quic.txt`, se connecte à `f5quic.com` sur le port 4433 et récupère le fichier `index.html` (qui n'existe pas mais peu importe, une réponse 404 nous suffit).

Parmi les messages affichés, vous verrez :

```
No token file present. Will create one as <demo_token_store.bin>.
```

Ces jetons ("*tokens*") sont définis dans le RFC 9000[1], section 8.1.1. Générés par le serveur, ils servent à s'assurer que le client reçoit bien les réponses, donc qu'il n'a pas usurpé son adresse IP <https://www.bortzmeyer.org/usurpation-adresse-ip.html>. Ce test de retournabilité <https://www.bortzmeyer.org/returnability.html> permet de se protéger contre les imposteurs. Comme c'est la première fois que nous lançons le client, il est normal qu'il n'ait pas encore de jeton.

Le fichier de résultat est téléchargeable ici, . Décortiquons-le :

```
efed1e171949e7a6: Sending ALPN list (8): h3-32, hq-32, h3-31, hq-31, h3-29, hq-29, h3-30, hq-30
```

La chaîne de caractères `efed1e171949e7a6` est le "*Connection ID*" (RFC 9000, section 5.1). QUIC utilise ALPN (RFC 7301) pour indiquer l'application à lancer à l'autre bout. `h3-32` est HTTP/3, "*draft version 32*" (dans le RFC final, le RFC 9113, l'ALPN est `h3`).

```
efed1e171949e7a6: Sending transport parameter TLS extension (89 bytes):
```

Diverses extensions au protocole TLS comme `Extension type: 9 (max_streams_uni), length 2, 4201` qui indique le nombre maximum de ruisseaux QUIC dans la connexion. Rappelez-vous que QUIC fusionne la traditionnelle couche transport et le chiffrement. Les paramètres de transport sont donc envoyés sous forme d'extensions TLS (RFC 9001, section 8.2). Ces paramètres figurent dans un registre IANA <https://www.iana.org/assignments/quic/quic.xml#quic-transport)>. Vous pouvez y vérifier que 9 est `initial_max_streams_uni`.

```
efed1e171949e7a6: Sending packet type: 2 (initial), S0, Q1, Version ff000020,
```

Les choses sérieuses démarrent. Le client envoie un paquet de type `Initial` marquant son désir d'ouvrir une connexion QUIC. Les types de paquets ne sont **pas** dans un registre IANA, leur liste limitative figure dans le RFC 9000, notamment dans la section 17.2. La version de QUIC désirée est `ff000020`, ce qui était la dernière version de développement. Aujourd'hui, ce serait `Version 1`. (La liste des versions, pour l'instant réduite, est dans un registre IANA <https://www.iana.org/assignments/quic/quic.xml#quic-versions>.)

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc9000.txt`

```
efed1e171949e7a6:     Crypto HS frame, offset 0, length 335: 0100014b030372b6...
efed1e171949e7a6:     padding, 867 bytes
```

Ce premier paquet contient une trame QUIC de type `CRYPTO` (RFC 9000, section 19.6), avec les paramètres cryptographiques, et du remplissage, pour atteindre la taille miminum imposée de 1 200 octets (RFC 9000, section 8.1)

```
efed1e171949e7a6: Sending 1252 bytes to 40.112.191.60:4433 at T=0.002156 (5bc54cac22aa1)
efed1e171949e7a6: Receiving 1200 bytes from 40.112.191.60:4433 at T=0.169086 (5bc54cac4b6b3)
```

Parfait, on a envoyé le paquet, et reçu une réponse. Petit rappel : l'information de base dans QUIC est transmise dans des **trames**, les trames voyagent dans un **paquet**, les paquets sont placés dans des datagrammes UDP. Il peut y avoir plusieurs paquets dans un datagramme et plusieurs trames dans un paquet. Déchiffrons maintenant la réponse :

```
efed1e171949e7a6: Receiving packet type: 2 (initial), S0, Q1, Version ff000020,
efed1e171949e7a6:     <b3efa80036784b1f>, <041eb19906b9ca99>, Seq: 0, pl: 149
...
efed1e171949e7a6:     ACK (nb=0), 0
efed1e171949e7a6:     Crypto HS frame, offset 0, length 123: 02000077030314b7...
```

Le serveur a répondu avec son propre paquet `Initial` (cf. RFC 9000, section 7, figure 4). Ensuite, il nous serre la main :

```
efed1e171949e7a6: Receiving packet type: 4 (handshake), S0, Q1, Version ff000020,
efed1e171949e7a6:     Crypto HS frame, offset 0, length 979: 0800006400620010...
efed1e171949e7a6: Received transport parameter TLS extension (82 bytes):
...
```

On est d'accord, on envoie nous aussi un `Handshake` :

```
efed1e171949e7a6: Sending packet type: 4 (handshake), S0, Q0, Version ff000020,
```

(Il y a en fait plusieurs paquets `Handshake`.)

```
efed1e171949e7a6: Receiving packet type: 6 (1rtt protected), S0, Q0,
...
efed1e171949e7a6: Sending packet type: 6 (1rtt protected), S1, Q0,
efed1e171949e7a6:     Prepared 1414 bytes
efed1e171949e7a6:     ping, 1 bytes
efed1e171949e7a6:     padding, 1413 bytes
```

Cette fois, la connexion est établie. On peut maintenant échanger des paquets de type `1-RTT`, le type de paquet général, qui sert pour tout sauf au début de la connexion. `ping` et `padding` sont des types de trames (RFC 9000, section 19). Les différents types de trames sont dans un registre IANA <`https://www.iana.org/assignments/quic/quic.xml#quic-frame-types`>. `ping` sert à déclencher l'envoi d'un accusé de réception, qui montrera que l'autre machine est bien vivante. `padding` fait du remplissage pour gêner l'analyse de trafic. Maintenant, c'est bon, le travail est fait, on peut raccrocher :

```
efed1e171949e7a6: Sending packet type: 6 (1rtt protected), S0, Q0,
efed1e171949e7a6:     <041eb19906b9ca99>, Seq: 4 (4), Phi: 0,
efed1e171949e7a6:     Prepared 8 bytes
efed1e171949e7a6:     ACK (nb=0), 0-4
efed1e171949e7a6:     application_close, Error 0x0000, Reason length 0

efed1e171949e7a6: Sending 34 bytes to 40.112.191.60:4433 at T=0.334640 (5bc54cac73d65)
efed1e171949e7a6: Receiving 37 bytes from 40.112.191.60:4433 at T=0.489651 (5bc54cac99ae8)
efed1e171949e7a6: Receiving packet type: 6 (1rtt protected), S0, Q1,
efed1e171949e7a6:     <b3efa80036784b1f>, Seq: 5 (5), Phi: 0,
efed1e171949e7a6:     Decrypted 11 bytes
efed1e171949e7a6:     application_close, Error 0x0000, Reason length 8
efed1e171949e7a6:         Reason: No error
```

Tout va bien, on a envoyé un paquet de type *"1-RTT"* qui contient l'alerte TLS `application_close`, la connexion est fermée.

Ici, on n'avait demandé qu'un seul fichier (`index.html`). Mais l'usage normal du Web est de demander plusieurs ressources, puisque l'affichage d'une page nécessite en général de nombreuses ressources (CSS, JavaScript, images...). Avec QUIC, ces différentes demandes peuvent être faites sur des ruisseaux différents, pour un maximum de parallélisme :

```
% ./picoquicdemo -l /tmp/quic.txt f5quic.com 4433 '0:index.html;100:foobar.html'
...
Opening stream 0 to GET /index.html
Opening stream 100 to GET /foobar.html
...
```

Et dans le fichier contenant les détails (), on va trouver cette fois des messages concernant ce ruisseau n° 100 :

```
7f0a5c80d9f66dab: Sending packet type: 6 (1rtt protected), S0, Q1,
7f0a5c80d9f66dab:     Stream 100, offset 0, length 32, fin = 1: 011e0000d1d7510c...
...
7f0a5c80d9f66dab: Receiving packet type: 6 (1rtt protected), S0, Q0,
7f0a5c80d9f66dab:     Stream 100, offset 0, length 245, fin = 0: 013e0000db5f4d91...
...
```

On a vu que la première connexion n'avait pas de jeton. Stocker un jeton reçu du serveur permet de faire du « zéro-RTT », c'est-à-dire d'envoyer des données dès le premier paquet transmis. Si on recommence la même commande `picoquicdemo`, le programme va lire le jeton précédemment obtenu, et stocké dans le fichier `demo_token_store.bin`, et s'en servir :

```
The session was properly resumed!
Zero RTT data is accepted!
```

Vous pouvez voir le résultat dans le fichier complet (). Après le paquet de type `Initial`, au lieu d'un type `Handshake`, on attaque directement avec un paquet de type `0-RTT` :

```
119d7339e68827df: Sending packet type: 5 (0rtt protected), S0, Q1, Version ff000020,
```

Et si on veut quand même utiliser l'excellent Wireshark, dont les versions les plus récentes savent décoder QUIC ? D'abord, sans truc particulier, on peut voir la partie non chiffrée des paquets QUIC. D'abord avec tshark, la version texte de Wireshark analysant un pcap pris en communiquant avec le serveur public de Microsoft (le pcap a été enregistré avec un `tcpdump -w /tmp/quic-microsoft.pcap host msquic.net and udp and port 443` :

```
% tshark  -r  /tmp/quic-microsoft.pcap
1   0.000000 10.168.234.1 → 138.91.188.147 QUIC 1294 Initial, DCID=9b4dccfc42f8ab9c, SCID=cf4d6e8c2c7b43a2, PKN
2   0.170410 138.91.188.147 → 10.168.234.1 QUIC 1294 Handshake, DCID=cf4d6e8c2c7b43a2, SCID=09fa3e7edeb3b621ae
3   0.170550 138.91.188.147 → 10.168.234.1 QUIC 1294 Handshake, DCID=cf4d6e8c2c7b43a2, SCID=09fa3e7edeb3b621ae
4   0.170558 138.91.188.147 → 10.168.234.1 QUIC 1108 Protected Payload (KP0), DCID=cf4d6e8c2c7b43a2
5   0.171689 10.168.234.1 → 138.91.188.147 QUIC 331 Protected Payload (KP0), DCID=09fa3e7edeb3b621ae
6   0.171765 10.168.234.1 → 138.91.188.147 QUIC 1514 Protected Payload (KP0), DCID=09fa3e7edeb3b621ae
```

On voit que les paquets d'établissement de la connexion, `Initial` et `Handshake` ne sont pas chiffrés (on n'a pas encore négocié les paramètres cryptographiques). On voit même que le paquet `Initial` contient deux trames, `CRYPTO` et `PADDING` (pour atteindre la taille minimale, imposée pour éviter les attaques avec amplification). tshark affiche les "*connections ID*" (DCID = destination et SCID = source). Une fois la cryptographie configurée, on ne voit plus grand'chose, la taille des paquets, et le "*connection ID*" de la destination (il s'agit de paquets à en-tête court, qui n'incluent pas le "*connection ID*" de la source). Même les paquets à la fin, qui terminent la connexion, ne sont pas décodables.

Voyons cette partie non chiffrée plus en détail avec l'option `-V` :

```
User Datagram Protocol, Src Port: 44873, Dst Port: 443
    Source Port: 44873
    Destination Port: 443
    Length: 1260
    UDP payload (1252 bytes)
QUIC IETF
    QUIC Connection information
        [Connection Number: 0]
    [Packet Length: 1252]
    1... .... = Header Form: Long Header (1)
    .1.. .... = Fixed Bit: True
    ..00 .... = Packet Type: Initial (0)
    .... 00.. = Reserved: 0
    .... ..11 = Packet Number Length: 4 bytes (3)
    Version: 1 (0x00000001)
    Destination Connection ID Length: 8
    Destination Connection ID: 9b4dccfc42f8ab9c
    Source Connection ID Length: 8
    Source Connection ID: cf4d6e8c2c7b43a2
    Token Length: 0
    Length: 1226
```

```
    Packet Number: 0
    Payload: 03c99cd0b086b40241f66768c1806ae00bcfa9f7db97593669e784fa326c83f89aac54d2...
    TLSv1.3 Record Layer: Handshake Protocol: Client Hello
        Frame Type: CRYPTO (0x0000000000000006)
        Offset: 0
        Length: 393
        Crypto Data
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Length: 389
    ...
            ALPN Next Protocol: h3
    ...
    Extension: quic_transport_parameters (len=85)
               Type: quic_transport_parameters (57)
...
               Parameter: initial_max_data (len=4) 1048576
                   Type: initial_max_data (0x04)
                   Length: 4
                   Value: 80100000
                   initial_max_data: 1048576
...
               Parameter: max_idle_timeout (len=4) 30000 ms
                   Type: max_idle_timeout (0x01)
                   Length: 4
                   Value: 80007530
                   max_idle_timeout: 30000
                ...
```

Il s'agit du paquet `Initial`, à en-tête long (le premier bit), incluant un classique "*Client Hello*" TLS, qui comprend notamment une demande ALPN pour HTTP/3, et les paramètres de transport spécifique à QUIC. La réponse du serveur inclut **deux** paquets QUIC (n'oubliez pas qu'un datagramme UDP peut transporter plusieurs paquets QUIC, chacun pouvant transporter plusieurs trames, ce qui est le cas du premier paquet, qui inclut une trame `ACK` et une trame `CRYPTO`) :

```
User Datagram Protocol, Src Port: 443, Dst Port: 44873
    Source Port: 443
    Destination Port: 44873
    Length: 1260
    UDP payload (1252 bytes)
QUIC IETF
    QUIC Connection information
        [Connection Number: 0]
    [Packet Length: 179]
    1... .... = Header Form: Long Header (1)
    .1.. .... = Fixed Bit: True
    ..00 .... = Packet Type: Initial (0)
    .... 00.. = Reserved: 0
    .... ..11 = Packet Number Length: 4 bytes (3)
    Version: 1 (0x00000001)
    Destination Connection ID Length: 8
    Destination Connection ID: cf4d6e8c2c7b43a2
    Source Connection ID Length: 9
    Source Connection ID: 09fa3e7edeb3b621ae
    Token Length: 0
    Length: 152
    Packet Number: 0
    ACK
        Frame Type: ACK (0x0000000000000002)
        Largest Acknowledged: 0
        ACK Delay: 16
        ACK Range Count: 0
        First ACK Range: 0
    TLSv1.3 Record Layer: Handshake Protocol: Server Hello
```

```
        Frame Type: CRYPTO (0x0000000000000006)
        Offset: 0
        Length: 123
        Crypto Data
        Handshake Protocol: Server Hello
...
QUIC IETF
    [Packet Length: 1073]
    1... .... = Header Form: Long Header (1)
    .1.. .... = Fixed Bit: True
    ..10 .... = Packet Type: Handshake (2)
    Version: 1 (0x00000001)
    Destination Connection ID Length: 8
    Destination Connection ID: cf4d6e8c2c7b43a2
    Source Connection ID Length: 9
    Source Connection ID: 09fa3e7edeb3b621ae
    Length: 1047
    [Expert Info (Warning/Decryption): Failed to create decryption context: Secrets are not available]
    ...
```

Le deuxième paquet, le `Handshake` est déjà chiffré.

Cette analyse peut aussi être faite avec l'interface graphique de Wireshark :

Maintenant, si on veut aller plus loin et fouiller dans la partie chiffrée des paquets ? Il faut pour cela que l'application écrive les clés utilisées pendant la connexion dans un fichier, et qu'on dise à Wireshark d'utiliser ce fichier pour déchiffrer. Si l'application utilise une bibliothèque qui permet d'inscrire les clés de session dans un fichier, comme OpenSSL ou picotls, on peut la convaincre d'écrire ces clés en définissant la variable d'environnement `SSLKEYLOGFILE`. Par exemple :

```
% export  SSLKEYLOGFILE=/tmp/quic.key

%  ./picoquicdemo  msquic.net  443  '0:index.html'

% cat /tmp/quic.key
SERVER_HANDSHAKE_TRAFFIC_SECRET 0469b5f648c6aece6b78338f4453f38d39a1527a5564458631a8d221c0d10ffa dc21c8a9927bc19
CLIENT_HANDSHAKE_TRAFFIC_SECRET 0469b5f648c6aece6b78338f4453f38d39a1527a5564458631a8d221c0d10ffa 53b7105c924b3f5
SERVER_TRAFFIC_SECRET_0 0469b5f648c6aece6b78338f4453f38d39a1527a5564458631a8d221c0d10ffa 683e037fb3e1d6451d1edce
CLIENT_TRAFFIC_SECRET_0 0469b5f648c6aece6b78338f4453f38d39a1527a5564458631a8d221c0d10ffa 2ca521c138a9e7ad66817b5
```

Il faut ensuite dire à Wireshark d'utiliser ces clés. Dans les menus Edit Preferences Protocols TLS, on indique le nom du fichier dans "*(Pre-)-Master-Secret log filename*". (Ou bien on édite le fichier de préférences de Wireshark, par exemple `/.config/wireshark/preferences` pour mettre `tls.keylog_file: /tmp/quic.key`.)

Désormais, Wireshark sait déchiffrer :

```
% tshark  -r  /tmp/quic-microsoft.pcap | more
1   0.000000 10.168.234.1 → 138.91.188.147 QUIC 1294 Initial, DCID=345d144296b90cff, SCID=f50587894e0cd26c, PKN
2   0.166722 138.91.188.147 → 10.168.234.1 HTTP3 1294 Protected Payload (KP0), DCID=f50587894e0cd26c, PKN: 2, ST
3   0.167897 10.168.234.1 → 138.91.188.147 QUIC 332 Protected Payload (KP0), DCID=d9e7940a80b4975407, PKN: 0, AC
4   0.167987 10.168.234.1 → 138.91.188.147 QUIC 1514 Protected Payload (KP0), DCID=d9e7940a80b4975407, PKN: 1, P
5   0.168550 10.168.234.1 → 138.91.188.147 HTTP3 225 Protected Payload (KP0), DCID=d9e7940a80b4975407, PKN: 2, S
), STREAM(10), STREAM(0), HEADERS, PADDING
6   0.332156 138.91.188.147 → 10.168.234.1 QUIC 1294 Protected Payload (KP0), DCID=f50587894e0cd26c, PKN: 3, ACK
7   0.332361 138.91.188.147 → 10.168.234.1 QUIC 1482 Protected Payload (KP0), DCID=f50587894e0cd26c, PKN: 4, PIN
8   0.332372 138.91.188.147 → 10.168.234.1 HTTP3 1294 Protected Payload (KP0), DCID=f50587894e0cd26c, PKN: 5, AC
9   0.332375 138.91.188.147 → 10.168.234.1 HTTP3 177 Protected Payload (KP0), DCID=f50587894e0cd26c, PKN: 6, STR
10   0.332841 10.168.234.1 → 138.91.188.147 QUIC 77 Protected Payload (KP0), DCID=d9e7940a80b4975407, PKN: 3, AC
11   0.487969 138.91.188.147 → 10.168.234.1 QUIC 80 Protected Payload (KP0), DCID=f50587894e0cd26c, PKN: 7, ACK,
```

On voit que la communication était utilisée pour HTTP/3 et on peut suivre les détails, comme les types de trame utilisés. Même chose avec l'option `-V`, on voit la totalité des paquets, ici un datagramme envoyé par le client :

```
User Datagram Protocol, Src Port: 56785, Dst Port: 443
    Source Port: 56785
    Destination Port: 443
    Length: 191
    UDP payload (183 bytes)
QUIC IETF
    QUIC Connection information
        [Connection Number: 0]
    [Packet Length: 183]
    QUIC Short Header DCID=d9e7940a80b4975407 PKN=2
        0... .... = Header Form: Short Header (0)
        .1.. .... = Fixed Bit: True
        ..1. .... = Spin Bit: True
        ...0 0... = Reserved: 0
        .... .0.. = Key Phase Bit: False
        .... ..00 = Packet Number Length: 1 bytes (0)
        Destination Connection ID: d9e7940a80b4975407
        Packet Number: 2
    STREAM id=2 fin=0 off=0 len=7 uni=1
        Frame Type: STREAM (0x000000000000000a)
            .... ...0 = Fin: False
            .... ..1. = Len(gth): True
            .... .0.. = Off(set): False
        Stream ID: 2
        Length: 7
    STREAM id=6 fin=0 off=0 len=1 uni=1
        Frame Type: STREAM (0x000000000000000a)
            .... ...0 = Fin: False
            .... ..1. = Len(gth): True
            .... .0.. = Off(set): False
        Stream ID: 6
        Length: 1
    STREAM id=10 fin=0 off=0 len=1 uni=1
        Frame Type: STREAM (0x000000000000000a)
            .... ...0 = Fin: False
            .... ..1. = Len(gth): True
            .... .0.. = Off(set): False
        Stream ID: 10
        Length: 1
    STREAM id=0 fin=1 off=0 len=31 uni=0
        Frame Type: STREAM (0x000000000000000b)
            .... ...1 = Fin: True
            .... ..1. = Len(gth): True
            .... .0.. = Off(set): False
        Stream ID: 0
        Length: 31
    PADDING Length: 104
        Frame Type: PADDING (0x0000000000000000)
        [Padding Length: 104]
...
Hypertext Transfer Protocol Version 3
    Type: HEADERS (0x0000000000000001)
    Length: 29
    Frame Payload: 0000d1d7510b2f696e6465782e68746d6c500a6d73717569632e6e6574
```

Le datagramme comprend un seul paquet, qui contient cinq trames, quatre de données (type `STREAM`) et une de remplissage (type `PADDING`). Dans les données se trouve une requête HTTP/3.

Graphiquement, on voit aussi davantage de détails :

https://www.bortzmeyer.org/quic-demo.html

Si vous voulez regarder vous-même avec Wireshark, et que vous n'avez pas de client QUIC pour créer du trafic, les fichiers (en ligne sur `https://www.bortzmeyer.org/files/quic-microsoft.pcap`) et (en ligne sur `https://www.bortzmeyer.org/files/quic.key`) sont disponibles.

`https://www.bortzmeyer.org/quic-demo.html`