EE542 - Reading Assignment – 08

The QUIC Transport Protocol: Design and Internet-Scale Deployment

Presenter: Boyang Xiao

USC id: 3326-7302-74

Email: boyangxi@usc.edu

Index

- Introductions and motivation
- QUIC design and implementation
- Experimentation framework
- Internet-scale deployment
- QUIC performance
- Experiences with QUIC and conclusions
- Q&A

Introductions and motivation

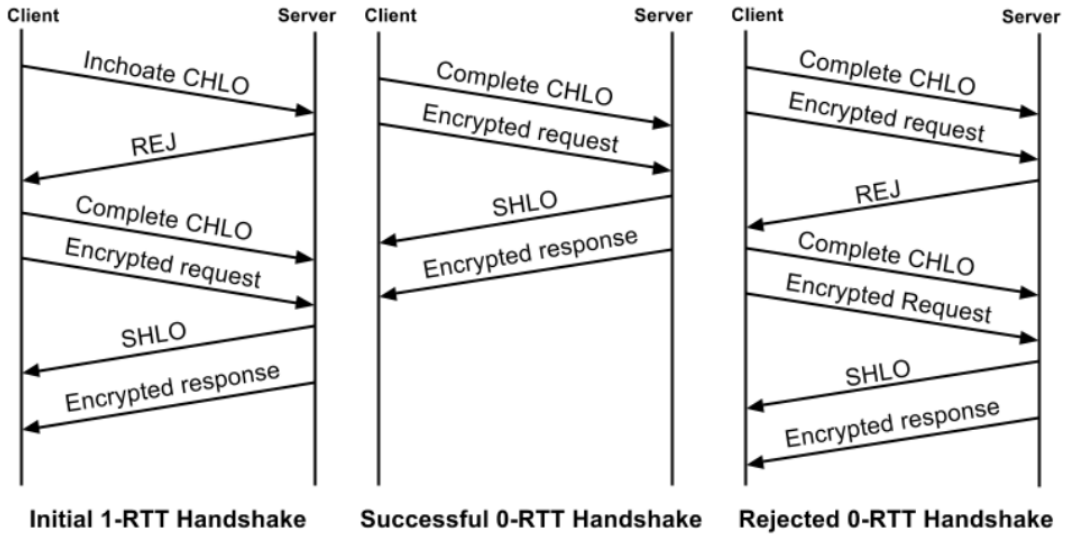
- QUIC overview
  - QUIC is an encrypted, multiplexed, and low-latency transport protocol designed from the ground up to improve transport performance for HTTPS traffic and to enable rapid deployment and continued evolution of transport mechanisms.
  - QUIC replaces most of the traditional HTTPS stack: HTTP/2, TLS, and TCP
  - QUIC uses a cryptographic handshake that minimizes handshake latency for most connections by using known server credentials on repeat connections and by removing redundant handshake-overhead at multiple layers in the network stack
  - QUIC eliminates head-of-line blocking delays by using a lightweight data-structuring abstraction, streams, which are multiplexed within a single connection so that loss of a single packet blocks only streams with data in that packet.

Introductions and motivation

- Why QUIC
  - **Protocol Entrenchment**: While new transport protocols have been specified to meet evolving application demands beyond TCP's simple service, they have not seen wide deployment.
  - **Implementation Entrenchment**: As the Internet continues to evolve and as attacks on various parts of the infrastructure remain a threat, there is a need to be able to deploy changes to clients rapidly
  - **Handshake Delay**: The generality of TCP and TLS continues to serve Internet evolution well, but the costs of layering have become increasingly visible with increasing latency demands on the HTTPS stack
  - **Head-of-line Blocking Delay:** TCP's bytestream abstraction prevents applications from controlling the framing of their communications and imposes a "latency tax" on application frames whose delivery must wait for retransmissions of previously lost TCP segments.

QUIC design and implementation

- Connection Establishment
  - **Initial handshake**: Initially, the client has no information about the server and so, before a handshake can be attempted, the client sends an inchoate client hello (CHLO) message to the server to elicit a reject (REJ) message.
  - **Final handshake**: All keys for a connection are established using Diffie-Hellman. After sending a complete CHLO, the client is in possession of initial keys for the connection since it can calculate the shared value from the server's long-term Diffie-Hellman public value and its own ephemeral Diffie-Hellman private key. At this point, the client is free to start sending application data to the server



QUIC design and implementation

- Stream Multiplexing
  - To avoid head-of-line blocking due to TCP's sequential delivery, QUIC supports multiple streams within a connection, ensuring that a lost UDP packet only impacts those streams whose data was carried in that packet.
  - A QUIC packet is composed of a common header followed by one or more frames. QUIC stream multiplexing is implemented by encapsulating stream data in one or more stream frames, and a single QUIC packet can carry stream frames from multiple streams.
- Authentication and Encryption
  - With the exception of a few early handshake packets and reset packets, QUIC packets are fully authenticated and mostly encrypted
  - The parts of the QUIC packet header outside the cover of encryption are required either for routing or for decrypting the packet: Flags, Connection ID, Version Number, Diversification Nonce, and Packet Number

QUIC design and implementation

- Loss Recovery
  - QUIC acknowledgments explicitly encode the delay between the receipt of a packet and its acknowledgment being sent. Together with monotonically-increasing packet numbers, this allows for precise network round-trip time (RTT) estimation, which aids in loss detection.
  - These differences between QUIC and TCP allowed us to build simpler and more effective mechanisms for QUIC.
- Flow Control
  - QUIC employs credit-based flow-control. A QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data. As data is sent, received, and delivered on a particular stream, the receiver periodically sends window update frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream.

QUIC design and implementation

- Congestion Control
  - The QUIC protocol does not rely on a specific congestion control algorithm. TCP and QUIC both use Cubic as the congestion controller, with one difference worth noting.
  - Since the audio and video streams are sent over two streams in a single QUIC connection, QUIC uses a variant of mulTCP for Cubic during the congestion avoidance phase to attain parity in flow-fairness with the use of TCP
- NAT Rebinding and Connection Migration
  - QUIC connections are identified by a 64-bit Connection ID. QUIC's Connection ID enables connections to survive changes to the client's IP and port.
  - While QUIC endpoints simply elide the problem of NAT rebinding by using the Connection ID to identify connections, client-initiated connection migration is a work in progress with limited deployment at this point.

QUIC design and implementation

- QUIC Discovery for HTTPS
  - When our client makes an HTTP request to an origin for the first time, it sends the request over TLS/TCP.
  - Our servers advertise QUIC support by including an "Alt-Svc" header in their HTTP responses. This header tells a client that connections to the origin may be attempted using QUIC. The client can now attempt to use QUIC in subsequent requests to the same origin.
- Open-Source Implementation
  - The source code is in C++, and includes substantial unit and end-to-end testing. The implementation includes a test server and a test client which can be used for experimentation, but are not tuned for production-level performance.

Experimentation framework

- QUIC experimentation is driven by implementing it in Chrome. Chrome's experimentation framework pseudo-randomly assigns clients to experiments and exports a wide range of metrics, from HTTP error rates to transport handshake latency. This framework also allows us to rapidly disable any experiment, thus protecting users from problematic experiments.
- QUIC support is also added to the mobile video (YouTube) and search (Google Search) apps as well.
- Google's server fleet consists of thousands of machines distributed globally, within data centers as well as within ISP networks. These front-end servers terminate incoming TLS/TCP and QUIC connections for all our services and perform load-balancing across internal application servers

Internet-scale deployment

- The process of deployment
  - QUIC support was added to Chrome in June 2013. In early 2014, we were confident in QUIC's stability and turned it on via Chrome's experimentation framework for a tiny fraction (< 0.025%) of users. As of January 2017, QUIC is turned on for almost6 all users of Chrome and the Android YouTube app.
  - In December 2015, Google discovered a vulnerability in the implementation of the QUIC handshake. The bug was fixed and QUIC traffic was restored as updated clients were rolled out.
  - The YouTube app started using QUIC in September 2016, doubling the percentage of Google's egress traffic over QUIC, from 15% to over 30%.
- Monitoring Metrics: Search Latency
  - Search Latency is defined as the delay between when a user enters a search term into the client and when all the search-result content is generated and delivered to the client, including images and embedded content.
  - The server infrastructure gathers performance data exported by frontend servers and aggregates them with service-specific metrics gathered by the server and clients, to provide visualizations and alerts.

QUIC performance

- Search Latency
  - Users in QUIC experienced reduced mean Search Latency. The percentile data shows that QUIC's improvements increase as base Search Latency increases.
  - Most of the latency reduction comes from the 0-RTT handshake: about 88% of QUIC connections from desktop achieve a 0-RTT handshake, which is at least a 2-RTT latency saving over TLS/TCP. The remaining QUIC connections still benefit from a 1-RTT handshake.
  - Search Latency gains on mobile are lower than gains on desktop.
- Video Latency
  - QUIC benefits mobile playbacks less than desktop. The YouTube app achieves a 0-RTT handshake for only 65% of QUIC connections.
  - The app tries to hide handshake costs, by establishing connections to the video server in the background while users are browsing and searching for videos.

QUIC performance

- Video Rebuffer Rate
  - Users in QUIC experience reduced Rebuffer Rate on average and substantial reductions at higher percentiles.
  - Rebuffer Rate for video playbacks as a function of the client's minimum RTT to the video server. However, QUIC's rebuffer rate increases more slowly than TCP's, implying that QUIC's loss-recovery mechanisms are more resilient to greater losses than TCP
  - Rebuffer rates can be decreased by reducing video quality, but QUIC playbacks show improved video quality as well as a decrease in rebuffers. QUIC's benefits are higher whenever congestion, loss, and RTTs are higher. As a result, we would expect QUIC to benefit users most in parts of the world where congestion, loss, and RTTs are highest.

QUIC performance

- Performance By Region
  - QUIC's performance benefits over TLS/TCP are thus not uniformly distributed across geography or network quality: benefits are greater in networks and regions that have higher average RTT and higher network loss.
- Server CPU Utilization
  - QUIC's server CPU-utilization was initially about 3.5 times higher than TLS/TCP.
  - The three major sources of QUIC's CPU cost were: cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state.
  - After optimizations, the CPU cost of serving web traffic over QUIC is decreased to approximately twice that of TLS/TCP, which has allowed us to increase the levels of QUIC traffic we serve.

QUIC performance

- Performance Limitations
  - **Pre-warmed connections:** When applications hide handshake latency by performing handshakes proactively, these applications receive no measurable benefit from QUIC's 0-RTT handshake.
  - **High bandwidth, low-delay, low-loss networks:** The use of QUIC on networks with plentiful bandwidth, low delay, and low loss rate, shows little gain and occasionally negative performance impact.
  - **Mobile devices:** QUIC's gains for mobile users are generally more modest than gains for desktop users.

Experiences with QUIC and conclusions

- Packet Size Considerations
  - The rapid increase in unreachability after 1450 bytes is a result of the total packet size— QUIC payload combined with UDP and IP headers—exceeding the 1500 byte Ethernet MTU.
  - Based on this data, 1350 bytes is chosen as the default payload size for QUIC
- UDP Blockage and Throttling
  - QUIC is successfully used for 95.3% of video clients attempting to use QUIC. 4.4% of clients are unable to use QUIC, meaning that QUIC or UDP is blocked or the path's MTU is too small.
  - We detect rate limiting as substantially elevated packet loss rate and decreased bandwidth at peak times of day, when traffic is high. We manually disable QUIC at our servers for entire Autonomous Systems (AS) where such throttling is detected and reach out to the operators running the network, asking them to either remove or at least raise their limits.

Experiences with QUIC and conclusions

- Forward Error Correction
  - Forward Error Correction (FEC) uses redundancy in the sent data stream to allow a receiver to recover lost packets without an explicit retransmission.
  - While retransmission rates decreased measurably, FEC had statistically insignificant impact on Search Latency and increased both Video Latency and Video Rebuffer Rate for video playbacks
  - The benefit of using an FEC scheme that recovers from a single packet loss is limited to under 30% of loss episodes
- User-space Development
  - We used a network simulator built into the QUIC code to perform fine-grained congestion control testing. Such facilities, which are often limited in kernel development environments, frequently caught significant bugs prior to deployment and live experimentation.

Experiences with QUIC and conclusions

- Experiences with Middleboxes
  - We identified the middlebox and reached out to the vendor. The vendor addressed the issue by updating their classifier to allow the variations seen in the flags. This fix was rolled out to their customers over the following month.
  - When traffic patterns change, they build responses to these observed changes. This pattern of behavior exposes a "deployment impossibility cycle" however, since deploying a protocol change widely requires it to work through a huge range of middleboxes, but middleboxes only change behavior in response to wide deployment of the change.
  - When deploying end-to-end changes, encryption is the only means available to ensure that bits that ought not be used by a middlebox are in fact not used by one.

Q&A

- Question: How does a client initialize QUIC communications with a server when sending HTTPs requests?

- Answer:
  - The client will first send a HTTPs request to the remote server over TLS/TCP. If the server attempts to establish QUIC communications with the client, it will add an "Alt-Svc" header in their HTTP responses. The client will then use QUIC to communicate with the server in the subsequent HTTPs requests.

Thanks for watching!

Presenter: Boyang Xiao

USC id: 3326-7302-74

Email: boyangxi@usc.edu