QUIC

Quick UDP Internet Connections

MULTIPLEXED STREAM TRANSPORT OVER UDP

Jim Roskind <jar@google.com>

To paraphrase a famous quote: *I apologize in advance for the length of this document, and if I had more time, I would surely have written less.*

First Draft 2012-04
Revised: 2013-06-24

Table of Contents

OVERVIEW

This is a working document, for group discussion and editing, which we expect to evolve into a somewhat fleshed out design document. The expectation is that we will flesh out a design for a tunneling protocol, running atop UDP, which can multiplex a large number of streams between two endpoints (a client, which initiates the overall connection, and a server). Each stream may, for example, be nearly equivalent to an independent TCP connection.

The eventual protocol may likely strongly resemble SCTP, using encryption strongly resembling DTLS, running atop UDP. Having an agreed list of goals and motivations should allow us to determine how much of such near-standards can be integrated, and where innovation and variations on such themes is needed, or useful.

MOTIVATIONS

We wish to reduce latency throughout the Internet, providing a more responsive set of user interactions. Over time, bandwidth throughout the world will grow, but round trip times, governed by the speed of light, will not diminish. We need a protocol to move requests, responses, and interactions through the internet with less latency along with fewer time-consuming retransmits, and we believe that current approaches are holding us all back. This section points at underlying issues that we wish to resolve.

Preamble: TCP and TLS (SSL) are excellent protocols, delivering remarkable results. This section will describe issues and shortcomings for our specific applications, which should not be viewed as a complaint about those remarkable pieces of work.

Pairs of IP addresses and sockets are finite resources. Today, too many distinct connections are routinely made between clients and servers, utilizing a multitude of sockets, and often carrying redundant information. A multiplexed transport has the potential for unifying the traffic and reducing the port utilization count. A multiplexed transport can unify reporting and responses to channel characteristics (packet loss, etc.), and also allow higher level application protocols (such as SPDY) to reduce or compress redundant data transmissions (such as headers). In the presence of layer-3 load-balancers, a multiplexed transport has the potential to allow the different data flows, coming and going to a client, to be served on a single server.

The two primary support motivations for this transport are to better support SPDY, and to further coalesce traffic. SPDY currently runs over TCP, but has encountered a few performance limitations.

SPDY SUPPORT MOTIVATIONS

SPDY is a multiplexed stream protocol currently implemented over TCP (routinely employing SSL). Among other things, it can reduce latency by sending all requests as soon as possible (not waiting for previous GETs to complete), and can reduce bandwidth utilization by compressing out some redundant traffic. Despite its features and successes, it has encountered several problems in its quest to be efficient in its use of resources while providing a latency reduction.

a) Single packet delay induces head-of-line blocking for a stream. Since TCP only provides a single serialized stream interface, a delay of only one packet causes the entire set of SPDY streams to pause. A packet is routinely delayed when a packet is lost, such as due to congestion, and it must be retransmitted. A better multiplexed transport should delay only one stream when a single packet is lost.

b) Unfavorable congestion avoidance handling by TCP, leading to additional bandwidth reduction and serialization latency overhead: A single SPDY connection is routinely used to replace K separate (non-multiplexed) connections. When a single packet of a SPDY (over TCP) connection is lost, the congestion window for the entire connection was historically reduced by 50%, courtesy of TCP [TCP CUBIC default congestion window reduction is roughly 30%, but the factor of 2 simplifies the math in this paragraph's exposition]. In contrast, a single packet loss among K non-multiplexed TCP connections only reduces the congestion window in one TCP connection. With only one of K streams impacted by a loss, the aggregate congestion window is reduced by roughly 1/2K of the pre-loss aggregate. With K commonly having a value of 6 (for multiple HTTP GET requests), a single packet loss causes bandwidth reduction to 11/12 of the pre-loss bandwidth (pre-loss congestion window size). As a result, TCP congestion avoidance favors sharded (multiple) TCP connections over a multiplexed TCP connection.

c) TLS (SSL) session resumption delays. A TLS handshake takes at least one additional RTT before data can be passed successfully. This delay is a function of the implementation of TLS, and not due to functional requirements of security.

d) TLS historically induced a decryption dependency, where prior packets must be

decrypted before later packets can be decrypted. TLS 1.1 and DTLS predominantly resolved the in-order dependency by adding explicit initialization vectors, at a cost of additional bytes per packet. By combining layers in a new protocol, we may be able to provide initialization vectors with reduced data costs, by leveraging other packetized data (e.g., packet sequence numbers).

GOALS

We'd like to develop a transport that supports the following goals:

1. Widespread deployability in ***today's*** internet (i.e., makes it through middle-boxes; runs on common user client machines without kernel changes, or elevated privileges)
2. Reduced head-of-line blocking due to packet loss (losing one packet will not generally impair other multiplexed streams)
3. Low latency (minimal round-trip costs, both during setup/resumption, and in response to packet loss)
    a. Significantly reduced connection startup latency (Commonly zero RTT connection, cryptographic hello, and initial request(s))
    b. Attempt to use Forward Error Correcting (FEC) codes to reduce retransmission latency after packet loss.
4. Improved support for mobile, in terms of latency and efficiency (as opposed to TCP connections which are torn down during radio shutdowns)
5. Congestion avoidance support comparable to, and friendly to, TCP (unified across multiplexed streams)
    a. Individual stream flow control, to prevent a stream with a fast source and slow sink from flooding memory at receiver end, and allow back-pressure to appear at the send end.
6. Privacy assurances comparable to TLS (without requiring in-order transport or in-order decryption)
7. Reliable and safe resource requirements scaling, both server-side and client-side (including reasonable buffer management and aids to avoid facilitating DoS magnification attacks)
8. Reduced bandwidth consumption and increased channel status responsiveness (via unified signaling of channel status across all multiplexed streams)
9. Reduced packet-count, if not in conflict with other goals.
10. Support reliable transport for multiplexed streams (can simulate TCP on the multiplexed streams)
11. Efficient demux-mux properties for proxies, if not in conflict with other goals.

12. Reuse, or evolve, existing protocols at any point where it is plausible to do so, without sacrificing our stated goals (e.g., consider uTP(Ledbat), DCCP, TCP minion)

JUSTIFICATIONS AND SOME IMPLICATIONS

The number one goal of viability ***today*** is clearly a major driver for this protocol development. With the understanding that middleboxes and firewalls would typically block or dramatically degrade any transport based on formats other than TCP or UDP, we will not even consider revolutionary protocols.

All parties would have preferred to satisfy goal 2 (re: congestion based packet loss on one stream, impacting several streams) by evolving TCP to avoid head-of-line blocking, but we found there was no apparent way to circumvent TCP's in-order delivery interface. Modifications are plausible, but could not be widely deployed until kernel changes were in place, and that was viewed as a showstopper. In addition, more significant modifications of TCP would potentially be blocked by today's middleboxes, which again would take many years to evolve.

Reduced latency in the face of packet loss (goal 3), as compared with TCP, may potentially be achieved by predominantly using error correction, rather than retransmission. Here again, such an extension of TCP did not appear plausible, without major (slow to deploy, perhaps over 10 years or more) modifications of standard TCP. We see wide deployment of this protocol as being plausible within 1-2 years, and would hope that TCP might evolve similarly over a much longer time period.

With the rise of mobile clients, and their tendency to turn off a radio communications, minimal round-trip latency costs (goal 4) for session continuations (resumptions?) will be progressively critical. Already today, desktop users benefit greatly from more rapid session continuations, and we're hoping that QUIC will take this to the ultimate goal of zero RTT for session resumption, including cryptographic negotiations (most of the time).

Congestion avoidance algorithms are critical to protecting the Internet. To the greatest extent possible we'll provide a TCP friendly and compatible protocol, to coexist fairly with TCP's highly evolved and universally deployed congestion avoidance approaches. We anticipate plausible improvements in agility and notification, courtesy of having a multitude of streams adjusted (re: congestion window?) in unison based on overall losses in all multiplexed streams.

Experience with SPDY development has taught us that the only way to prevent middleboxes from maligning a new protocol built atop UDP or TCP (e.g., misconstruing it for a “known” protocol, and making “less than helpful” changes), is to encrypt as much of the payload and control structure as feasible. As a result we plan to employ security elements including tamper resistance, privacy, replay protection, and authentication, that is similar to what is provided via TLS (perhaps DTLS like).

Compression will not be directly incorporated into the transport, beyond header compression comparable to SPDY, as the transport may not be aware of privacy/sensitivity restrictions between various segments of data, and must not reveal (via traffic analysis) similarities in what should be isolated groups of data. SPDY’s current “continuously adaptive” compression of headers is mildly problematic, as headers for “other streams” can’t be decompressed until all prior headers (which are fed into the evolving compression context) have been received. QUIC may initially support such SPDY-like header compression at the expense of head-of-line blocking among such compressed packets, but we should be able to experiment with variations that reduce this dependency. Given the effectiveness of SPDY-like header compression, it may often be the case that very few packets will actually contain headers, further mitigating the potential downside.

Privacy mechanisms will need to be incorporated to carefully align encryption blocks with packet boundaries, or at least redundancy protected byte ranges, so that the latency impact of packet loss is maximally contained. Note that if these boundaries are not well aligned, then a single packet loss may effectively preclude deciphering of adjacent packets, with an undesirable impact on latency.

WHY NOT USE SCTP over DTLS?

One recurring question is: Why not use SCTP (Stream Control Transmission Protocol) over DTLS (Datagram Transport Layer Security)? SCTP provides (among other things) stream multiplexing, and DTLS provides SSL quality encryption and authentication over a UDP stream. In addition to the fact that both of these protocols are in various levels of standardization, this combination is currently on a standard track, and described in this Network Working Group Internet Draft.

The largest visible issue with using these protocols relates to our goals in the area of connection latency, and is perhaps the most critical conflicting element. In addition, we can also anticipate issues in bandwidth efficiency that may reduce our ability to achieve the goals of QUIC.

CONNECTION ESTABLISHMENT LATENCY OF SCTP over DTLS

One of the major and believably achievable goals of QUIC, is to predominantly have zero RTT connection (re)establishment, as was mentioned in goal 3a above. It is highly doubtful that SCTP over DTLS can come close to achieving that goal.

The fact that SCTP and DTLS are currently implemented as layered protocols, one atop the other, already instigates an overall connection latency that is the sum of the two connection latencies. This is seen in section 5.1 of the Draft, where they note:

- A DTLS connection MUST be established before an SCTP association can be set up.

SCTP (alone) appears to require 1 full round trip in connection establishment prior to any data transfer. See section 5 of the SCTP RFC 4960 for a discussion of this requirement.

DTLS appears to routinely require 3 round trips in its connection establishment. DTLS is modeled after TLS, which defaults to a 2 RTT HELLO exchange at connection establishment. As noted in section 8.1 of DTLS's description: DTLS generally uses a full 3 round trips to negotiate a connection (including cookie exchange). That section notes that the 3 round trips are comparable to a TLS over TCP connection establishment, if the TCP's connection RTT is accumulated with the 2 RTTs for the TLS HELLO.

As a result, with the above baseline connection round trips, we would anticipate a cost of roughly 4 round trips to establish an SCTP over DTLS connection. In contrast, we expect to be able to perform a QUIC connection establishment with zero RTT overhead. Worst case for QUIC should be one RTT overhead utilizing approaches kindred to SSL False Start. We anticipate being able to reduce the probability of that False Start case to very low levels.

It is plausible that some of the connection overhead in SCTP over DTLS could be reduced, but it is highly doubtful that without major changes and merging of the protocols, that a competitive startup latency could be achieved, and the goals of QUIC met.

EFFICIENT UTILIZATION OF BANDWIDTH FOR SCTP over DTLS

The layering of SCTP over DTLS tends to make it more difficult to most efficiently utilize bandwidth in UDP packets. As an example, the SCTP transport conveys data to order packets, and data ranges. At the same time, DTLS is required to separately transport distinctive packet information in the form of cryptographic initialization vectors, used for

decryption of packets. Integrated protocols, such as QUIC, can utilize the sequencing information for the stream protocol, as the basis for the initialization vector in packet decryption. The result of this integration is reduced packet overhead. We anticipate that this integration will save data space in several additional places, notably increasing packing efficiency. [TBD: We should provide specific gains, and byte counts, per packet, to fully flesh out this argument.]

PACKET LOSS RETRANSMISSION LATENCY

Goal 3b suggests that we should attempt to reduce retransmission latency by utilizing FEC codes. Neither DTLS nor SCTP currently have provisions for using FEC, and it would appear to be a very significant modification to try to incorporate such. There are some efforts to add FEC beneath SCTP, so it is perhaps plausible, but it would certainly be a notable complication As currently defined, packet loss would definitely tend to induce retransmission latency in those protocols, although it would be restricted (courtesy of SCTP) to impacting a single stream (at least achieving a goal 2).

Expected API Elements

API CONCEPTS

There are several complexities to be ironed out in establishing an API for multiplexed streams. At the highest level, we need to have a mechanism that adds new streams to a connection, as well as separately reading and writing various streams independently.

For each stream, we need a way to access the stream, and to specify characteristics that should be employed for the stream. Characteristics include, for example, reliability, and performance tradeoffs (such as jitter reduction via added redundancy, vs. bandwidth reduction via reduced redundancy).

STREAM CHARACTERISTICS

We expect that different streams will have distinct transport characteristics which may be set or modified by the application. These include such distinct characteristic settings as:

- Adjustable redundancy levels (trade bandwidth for latency savings)
- Adjustable prioritization levels (modeled after SPDY's evolving prioritization schemes).

We expect that some control channel, which may be viewed as an out-of-band stream, will always be available and may be used to signal state changes for the rest of the streams.

The control channel will likely consist of special purpose frames (control frames), as well a reserved stream, for cryptographic negotiations.

IN-SEQUENCE DATA DELIVERY

Existing code will surely need services kindred to TCP's reliable, and in-order-delivery. As a result, we need an API for a stream that greatly mimics a standard TCP socket. This will allow a multiplexed stream to be experimentally used with almost any existing application, with hopefully trivial modifications. We expect this API to mimic the interfaces provided by SPDY, while masking internal packet loss, and providing data in each stream (in order) independent of the transmissions of other streams.

CONNECTION STATUS

The separation between an application and the actual connection has historically made the use of a connection difficult. For example, when a sending application is finished sending, it may try to close the connection, but the data may still be queued up locally and not yet sent (or not yet ACKed, and hence the send buffer cannot be discarded). Such examples create races which may lead to undefined behavior when closing a connection, or terminating an application.

To better support an efficient and tight binding with an application, the following current statistics are plausibly expected to be made visible to the application:

1. RTT (current smoothed estimate)
2. Packet size (including all overhead; also excluding overhead, only including payload)
3. Bandwidth (current smoothed estimate across of entire connection)
4. Peak Sustained Bandwidth (across entire connection)
5. Congestion window size (expressed in packets)
6. Queue size (packets that have been formed, but not yet emitted over a wire)
7. Bytes in queue
8. Per-stream queue size (either bytes per stream, or unsent packets, both??)

Notification should also be provided, or access for the following events [granularity of notification is TBD, and there should be no requirement on timeliness of the notifications, but any notification or status should include a best estimate of when the actual event took place]:

1. Queue size has dropped to zero

2. All required ACKs have been received (connection may be closed with no transmission state loss.)
    a. ACK of specific packet (section of stream?) has been received (not all streams support this. [Should this be queryable, rather than a notification??])

PROTOCOL PHILOSOPHY

The protocol has four phases where we need to consider performance efficiency: Startup; Steady State; Idle Entry; Idle Departure. One critical element is reducing latency during startup (connection establishment), especially in the case of resumption. The second challenge is to ensure efficient and low latency performance in a steady state, when a multitude of streams are keeping the overall connection as full as congestion avoidance will allow. The third challenge is to efficiently and with low latency ensure smooth transitions into an idle state, where no streams are currently supplying data, but the connection is fully established. That transition is notably where tail-drop (packet loss) in TCP has traditionally incurred significant latency, especially when a final packet that is lost, and a send-side timeout is required to re-send lost data.

CONNECTING VIA CONNECTIONLESS UDP: OVERCOMING NATs

A most fundamental issue is how to turn UDP datagrams, into a connection based protocol. This issue is exacerbated by middle boxes and firewall NAT services that do not assist, and actually can hinder the process.

As an example, consider the case when a middle box, such as a firewall or NAT, decides to no longer support a specific TCP connection. The middlebox can send a TCP RST (connection RESET) to both endpoints as part of a tear-down notification. In contrast, when a NAT service decides to discard a binding used by UDP traffic, there is no notification, as a UDP datagram is not expected to be a “connection.” Once a UDP NAT unbinding has taken place, the external endpoint (typically a server) is left with no means of sending traffic to the client. Worse yet, traffic from the server is typically black-holed, with no NACK response. With TCP, an attempt to use an unbound port may result in at least a RST, which is comparable to a NACK. As a further complication, if a NAT port unbinding takes place, and then a ***client*** sends additional UDP traffic, the NAT service may create a new binding. That new port binding may have a distinct source-port (as seen by the server), despite the fact that QUIC will need to view the traffic as a continuation of the existing connection.

Current estimates suggest that it is common for currently deployed NAT boxes to unbind a UDP port mapping after an idle time in the range of 30 to 120 seconds, and unbinding may take place sooner. Early unbinding may be caused by LRU NAT table eviction, or any of a variety of "weaker" implementations, such as pseudo-random hash table eviction. Both of these unbinding activities are problematic to our protocol, and must be handled well. In the future, it is possible that RFC 4787 may force minimal timeouts to 120 seconds, and default timeouts to 5 minutes, but that is all future conjectured acceptance and deployment, and we need to deal with today's middleboxes.

Most anticipated traffic between a client and a server is of the form of a client request, followed by a server response. That structure suggests that (premature) NAT unbinding might be less critical. Unfortunately, there are numerous situations where the server response is delayed for extended period of time, including server back-end response time. As a common and more extreme example, statistics gathered via Chrome show that roughly 0.5% of all HTTP GET connections receive no response until 60 seconds after the connection was opened. Such delay is presumed to be caused by "hanging GETs." Our protocol needs to be careful to handle such server-side pauses, and still allow a server to respond with (belated) data on an open stream.

One basic element of NAT translation takes place at a LAN boundary. Unless a client makes an outbound connection through a NAT, there is generally no way for a server to contact (respond to?) a client, unless services like UPnP are employed. Given that PCP (Port Control Protocol: RFC 6887) support is not universal, we will design the protocol to not depend on such services. We will, at the client side, check for the presence of this service, and use it when available. As we evolve QUIC, we need to run experiments to see how often UPnP or PCP is present, and how often it works.

GUID: THE KEY TO CONNECTION IDENTIFICATION

Given that NAT services can vary the WAN-visible source-port over the lifetime of a connection (via unbinding, and rebinding), it is clear that source-IP address and source-port are strictly insufficient to define a connection. We overcome any such confusion by using a GUID (Globally Unique IDentifier) which generally persists for the lifetime of the connection. A GUID is a pseudo randomly generated nonce [size is currently set at 64 bits], that is expected to be universally unique. The GUID is usually proposed in the first UDP packet sent by a client to a server, and is present explicitly or implicitly in all future packets that are exchanged for the lifetime of the connection. It is the defining key for the connection.

A server can use this GUID as a key to identify the specific connection from among many inbound connections sending UDP packets to a single server port. The GUID is also used by the server to resurrect the current session encryption context for use with AEAD (Authenticated Encryption with Associated Data). That context conceptually includes an encryption key, as well as an authentication key. (Note: mutable portions of the packet, such as source-port, etc., are not included in the Associated Data authentication).

NAT BINDING KEEP-ALIVE

As noted, NAT port bindings may be discarded without notice by a NAT service. The only remedy for such a discard is to send additional traffic from a client to the server, re-establishing an active NAT port mapping. A port binding must be proactively re-established if there is any chance that a server may attempt to send traffic to the client, and there is reason to believe that the existing port binding may have been discarded (timed out).

Proactive establishing of a port binding is expensive, as it is presumably done when there is no “real” traffic to send. For mobile devices, such keep-alive transmission may be especially problematic, as they can impact power consumption. As a result, it is desirable to minimize the frequency of keep-alive transmissions, as well as the absolute count of such keep-alives.

KEEP-ALIVES: WHEN ARE THEY NEEDED?
To simplify the question of when we need to keep the connection alive, we will assume that a server will only send traffic to a client while at least one multiplexed stream is open on the connection. Hence, if a connection is being used for spontaneous transmission from the server to the client, a stream must be kept open.

KEEP-ALIVES: HOW OFTEN ARE THEY NEEDED?
In some cases, inbound traffic from a server may be sufficient to keep-alive a NAT binding, but some NAT servers require an outbound packet from the client to reset the table entry expiration time. An outbound packet also serves to recreate a NAT binding if the old binding has expired, and hence can heal any problem that might be caused by mis-timings.

Other UDP based protocols, such as for audio and video streaming, routinely send UDP keep-alive packets roughly every 15 seconds. We need to experiment to determine baseline timeout, and adapt to the environment we sense.

In order to monitor the effectiveness of the keep-alives, the server will automatically send a

keep-alive probe after an algorithmically determined (or negotiated) timeout in idleness (since the last client packet transmission). The client will maintain a corresponding timer, and will be alerted when the time has expired, and some additional time has passed (e.g., additional time might be one or more estimated RTTs). The client will then either ACK that probe packet's arrival, or effectively NACK the expected probe packet shortly after it should have arrived. In either case, the client will send a packet, and ensure that the NAT binding is revitalized. As a result of this process, the endpoints will be able to shorten the timeout interval (after a NACK), or potentially lengthen the interval (after an ACK). That adaptive process should converge on the apparent NAT timeout, but can be clamped to not exceed some limits, so as not to provide a DOS vector to attackers.

The exact algorithm is TBD. The keep-alive timeout can also be negotiated (example: server may recognize a mobile network with a NAT service having a long time-to-live table), or may be persisted from previous connections. After some real world experimentation, we may decide that some clients, in some environments, are not good candidates for using QUIC, as their keep-alive requirements are not tolerable for one or both endpoints.

UDP PACKET FRAGMENTATION

UDP packets that are larger than the MTU on a single link are at risk of being fragmented at the IP level. It is TBD whether we'll mark all of our packets as "do not fragment," which would result in uniform loss of packets if the MTU is exceeded. This section discusses some of the pros and cons, and possible approaches. The most critical expected realization is that fragmentation in today's (and tomorrow's) internet is becoming progressively less and less common. As a result, this section may be of very little significance, and any/all decisions on this issue may prove workable.

When a packet is fragmented at the IP level, the initial packet will continue to hold the UDP source and target port specifiers, as well as a GUID, but all latter fragments will be devoid of such identifying information. The only means of reassembly will be the IP level "identification" field, which is only 16 bits in length. As a result, if more than (roughly) the square root of 2^16 packets are in transit at the same time, the probability of a collision (and broken reassembly) will be large. For example, when windows on connections exceed about 256 packets, there will probably be collisions.

Even if effective congestion windows are "small" (much less than 256 packets), a NAT router may direct a collection of separate connections toward a single server IP address (from the IP address of the router). That collection may easily have more than 256 packets

in flight at once, competing for unique IP identifiers, and instigating reassembly collisions (if fragmented).

Collisions, probably based on the identical MTU boundaries, will (in the face of packet reordering), be virtually impossible to reassemble correctly. As a result, the receiving party will either abandon the reassembly effort (on some fraction of the fragmented packets), or it will provide an erroneous reassembly. Packets that are mis-assembled will be detected as garbled by an authentication hash. As a result, reassembly errors cannot cause protocol errors that are any worse than discarding the packets that might be fragmented.

Fragmentation can impact bandwidth utilization, and can wastefully take time from a target server. Given that the reassembly conflicts are pretty much guaranteed to exist, if we have any significant fragmentation, it may be desirable to preclude fragmentation, and avoid wasting reassembly efforts at the server. On the other hand, if this issue is miniscule, it may be desirable to have slightly better coverage and support the intermittent fragmentations.

Currently, the OS API does not allow an application level code to detect the fact that IP fragmentation has taken place. We may seek to enhance our server OS support, so as to detect fragmentation. With such an API, we can then attempt to work within any visible MTU limitation. This approach would traditionally avoid error-prone “path MTU exploration,” and would instead observe and respond to actions taken on actual UDP packets. [TBD: An attacker might choose to intermittently fragment some packets, so as to cause us to negotiate down to a lower and less efficient MTU. We may find such a result unacceptable. If so, we could globally decline to adapt to MTU, or decline to facilitate transit or reassembly of fragmented packets.]

CONNECTION ESTABLISHMENT and RESUMPTION

To minimize latency at startup, and expedite data responses to a first contact, the very first packets sent over QUIC will often include session negotiation information as well as one or more requests. QUIC is designed to speculatively assume the client has acceptable cryptographic credentials needed for at least a preliminary encryption of a request, that it has sufficient connection credentials that an anti-DDOS challenge (round trip) is not needed, and that it has sufficient freshness proof that replay attacks can be precluded. If the server declines to accept the credentials, additional round-trip negotiations, comparable to TCP session establishment, or SSL hello negotiations, may ensue.

STARTUP DDOS ATTACKS

A known problem with rapid startup is denial of service attacks. In such attacks, a malicious client may provide a fallacious return (response) address, and may try to cause a server to expend significant computational resources, and/or send significant traffic to a third-party address. Historically, these attacks on TCP servers have been avoided by requiring an extra round trip (confirming a return address), and utilization of SYN cookies. Recent work to extend TCP with TCP Fast Open offers promising strategies for evolving TCP to include data in the initial SYN packet, with plausibly acceptable controls on DOS attacks.

QUIC will have to address the issue of expiration of connection credentials, and/or other mechanisms to preclude replay attacks using those credentials. We expect to design the protocol, including the cryptographic resumption, to be as robust as the proposed TCP Fast Open and handling such attacks. That paper includes both proof-of-ownership for a source IP (based on previous connection history), as well as automatic fallback to 3-way-handshake style connections (re: add an extra RTT) when a server is being attacked or appears overloaded. That fact that our protocol consistently includes end-to-end encryption can assure us that ownership credentials cannot be trivially stolen (via eavesdropping).

SECURITY CREDENTIALS

A client should retain information about a server from previous contacts with said server. Retained information should be at least sufficient to support the analog of TLS session resumption, and should also include server information, such as a server's most recently used public key (historically kept in a cert, with an associated chain to a trusted root). Clients should speculatively assume that the last known server public key(s) is still in use (unless there is evidence of revocation or replacement), and attempt to utilize this information to achieve a zero RTT transmission of encrypted initial payload transfer (requests?).

Support for encryption where the server doesn't have a round trip in which to add randomness into the connection requires that the server maintain state in order to avoid replay attacks. That state can be limited in time by assuming some amount of clock sync, and in space by giving the client an ID that identifies the shard (called an *orbit* in Snap Start). The server may be unable to establish the uniqueness of the connection, in which case the client will have to be prepared to re-encrypt and re-transmit everything that it optimistically sent.

Likewise, if the server's key has changed, the client will have to re-encrypt and re-transmit the optimistic, initial flow of data.

The initial flow in a zero-RTT setup is potentially less forward secure than possible so we should assume that the connection will upgrade to a truly ephemeral key for subsequent flows on the same connection. Although the default setting supporting 0-RTT connection establishment should be the default, it should be possible for a server to insist that only perfectly-forward-secure encryption be used for even the initial traffic.

HIGH LEVEL OVERVIEW OF CONNECTION SCENARIOS

This section is meant to provide an overview of how QUIC connections will typically proceed. We'll sketch this out emphasizing the number of RTTs that result in each case. We'll also provide the motivation for some of the details that are used to mitigate potential attacks.

In all cases, the server may select to force a fallback to a slower connection scenario when it is heavily loaded, or potentially under a DOS attack. In some cases, a server may select to speculatively risk being minimally victimized by an attack, especially when the cost of the attack is limited to expending computational resources server side, and the server is lightly loaded. The descriptions given below attempt to maximally protect third parties from reflection attacks during the connection formation. Other sections will address third party attacks during steady state, and roaming.

FIRST EVER CONNECTION: Usually 1 RTT, sometimes 2 RTTs

In this scenario, a client contacts the server, and its initial hello message indicates that the client has never before visited the server, and hence it cannot speculate about a public key. The initial message from the client may include some randomness that will expedite a session negotiation. The entire client message will fit in a single packet, and should pad/fill that packet. By filling the first client-generated packet, we are assured that a full-packet-response can be sent by the server, and that it will typically be no larger than the client's first packet. This full-packet will also raise the bandwidth cost for any attacker (supplying a false return address), diminishing the potential, if any, for any reflected amplification. At the same time, this full-packet is a negligible cost to an honest client, which should rarely have to use this connection strategy, and will shortly exchange data that will dwarf the few extra bytes that are used as padding in this first packet.

The server will respond with a server certificate, which can usually fit in a single packet.

Typical certificate sizes are 1-1.5KB, and for example Google's current mail.google.com certificate is about 806 bytes. In order to minimize the size of this server response, the server will only include the hashes of the certificate chain (rather than a complete list of certificates in a certificate chain). Minimizing this server response reduces (or eliminates) any reflected amplification potential. The hashes of the certificate chain are sent as a speculation that the client will likely be able to decode the hashes, and then validate the chain. In addition, the server includes in this response the moral equivalent to a SYN Cookie, which is a very-short-lived proof that the recipient controls the targeted client source IP and port. Note that it is very short lived, because it is sent in the clear, and could be misappropriated by a malicious third party.

When the client receives the above packet, it can attempt to validate the enclosed server certificate. The first step in validating the certificate is to decode the certificate chain hashes. If the client is unable to decode sufficiently the certificate chain hashes, or the certificate chain appears to not validate the server certificate, then the client enters into the 2 RTT case, and is forced to send a request to the server for the fully elaborated certificate chain. That request is sent along with the SYN Cookie, which allows the server to safely send the additional packets to the client (without risk of a misdirected amplification attack).

Eventually (after either 1 or 2 round trips), the client will then have both an authenticated server certificate, and also a short lived analog of a SYN Cookie. With that information, the client can proceed into the 0 RTT case described below, and be recognized instantly as the trusted owner of the client return IP address and port. It is possible that additional entropy may have been exchanged as well, that might expedite or allow cryptographic negotiations, but is not critical to this overview.

REPEAT CONNECTION: Usually 0 RTT; sometimes 1 RTT; rarely 2 RTT

In a repeat connection, a client will speculate that the server is still using the server certificate that was seen and (previously) validated. In addition, the client may be in possession of proof that it has control over its return IP address and port. Proof may include the analog of the SYN Cookie described in the "First Ever Connection" section above, or may include an analog of the TCP Fast Open cookie, which is longer lived (valid for a much longer period of time).

To proceed with this connection, a client will construct the makings of the analog of a TLS Session Master Secret for inclusion in a message. Techniques used in SSL Snap Start will be used for this construction, so as to mitigate replay attacks. Also included in the constructed (and encrypted) message will be the "proof" of control over the client's IP address, as well as any QUIC negotiable items, such as proposed congestion avoidance

algorithm, etc. The encryption of the proof (such as the analog of the TCP Fast Open cookie) will ensure that an eavesdropper cannot steal and abuse such proof.

The above cryptographic hello message will be sent to the server in a single packet. After sending that hello, the client can then encrypt, authenticate and send additional data packet(s), such as are required to open streams, make requests, etc. [To reduce the impact of packet loss, which could delay the entire connection establishment, those packets will potentially be sent more than once. For example, it is likely that the crypto hello packet would be sent, and then the data packets, then a resend of the crypto hello packet (possibly after a slight pacing delay?), then a resend of the data packets. TBD we may use some retransmission logic and timeouts that are set shorter than usual.]

When the receiver gets the cryptographic hello packet, it will evaluate its contents. If the speculated server public key is no longer in use, or the message's proposed negotiation results are not acceptable to the server, then the server may reject the contents of the packet, and treat it instead as kindred to the "First Ever Connected" case, described above. In that case, the data packets that later arrive utilizing the proposed Master Secret will be discarded (there is no way to decrypt them at the server side).

If the public key is still acceptable, and the hello message is acceptable, then the proof of control of the client IP and port is considered. In the common case, the proof of control of the client IP is sufficient, and the data packets that follow can be decrypted, processed, and acted upon (just as we do with an HTTP GET after a TCP channel has completed a SYN + ACK round trip).

PROOF OF OWNERSHIP OF CLIENT IP ADDRESS

There are several reasons for the proof of ownership of the client IP to be insufficient. Many of these reasons relate to how busy a server is, and how likely it is to be under a resource utilization attack, which may then raise or lower the bar for the proofs. Similarly, if the proof is older it may be deemed less trustworthy. Finally, if the proof is not directly applicable, but is a proof for a "nearby IP" (actual IP, vs. proof supplied IP), then it may be deemed more trustworthy than proof for a more distantly related IP. If the proof is not sufficiently trusted, then the server may send a rejection (probing) packet to assure that the client return IP and port is authentic.

GENERATING PROOF OF OWNERSHIP/CONTROL OF CLIENT IP ADDRESS

During a connection, a QUIC server may create and transmit a statement that a client is using a specific IP address and port, at a specific time. Such a statement will include a MAC generated by a server secret, so that the server can authenticate that statement as

proof in a 0-RTT “Repeat Connection.” A server may push updates of this statement (with more recent timestamps) during a connection.

A client may assemble a list of one or more such proofs for distinct IP addresses. For instance, it may attain a proof of ownership of an IP address used by a home Wi-Fi firewall, as well as an IP address used in a 3G mobile connection. In cases where roaming is plausible, or likely, a client may supply a server with several of the ownership proofs in anticipation of future roaming. When a server is made aware of possible roaming in advance (and of the specific IP addresses that may come into play), it can then typically immediately respond to a change of client return IP address, without delaying the transition to perform a probe.

STEADY STATE

Each stream within the QUIC connection will have a unique stream identifier associated with it.

Byte stream based data transport will be modeled after TCP, with byte ranges provided with payload data. Byte ranges will select positions within each byte stream.

CONNECTION STRUCTURE

Streams will be partitioned into frames for placement into (UDP) packets. Whenever possible, any particular packet's payload should come from only one stream. Such dedication will reduce the probability that a single packet loss will impede progress of more than one stream. When there is not sufficient data in a stream to fill a packet, then frames from more than one stream may be packed into a single packet. Such packing should reduce the packet-count-overhead, and diminish serialization latency.

SECURITY: TAMPER RESISTANCE, PRIVACY, AUTHENTICITY

Given a packet, we need a way to identify the cryptographic context. The context will be associated with a GUID present in each (UDP) packet.

Packets will be protected with AEAD (Authenticated Encryption with Associated Data) keyed on a shared secret held in the cryptographic context. We expect that roughly 8 bytes of authentication overhead will be needed in each packet, but the details are TBD based on the state of the art standards.

Each packet needs some initialization vector (IV) bytes of nonce for use in encrypting. The nonce needs to be unique for every packet. An AEAD algorithm will be selected for which the nonce can be a simple counter (i.e., must be unique, but predictability of the IV is acceptable), and will be based on the packet sequence number.

We should provide support to pad packets to reduce vulnerability to traffic analysis. At this point, anti-traffic-analysis is probably not sufficiently well understood to design countermeasures that use that ability.

The security protocol should be designed to minimize round trips. We have experience with zero RTT security protocols and we can do significantly better than (D)TLS.

ISOLATED-PACKET-ENCRYPTION

In any isolated-packet-encryption mode, we avoid having encryption add to latency beyond that which is mandated by a delayed packet. For example, if decryption of a packet required either the plaintext or ciphertext of some other packet (e.g., some cipher block chaining mode), then we would not have an isolated-packet-encryption property. Without that property, the delay (loss?) of a packet would delay the decoding of another packet, and be unacceptable.

PACKET LOSS

Packet loss in the Internet is broadly estimated to be in the range of 1-2% of all packets. These numbers have been confirmed by tests of clients, such as Chrome, recording stats for test streams of UDP packets to server farms around the world. The primary cause for packet loss is believed to be congestion, where routers perform switching operations, and output buffer sizes are exceeded. This issue is fundamental to the design of the Internet, and TCP, where packet loss is used as a signal of congestion, and the protocol is required to respond by reducing the flow across the congested path. Packet loss can also be caused by analog factors on transmission lines, but such losses are believed to be much lower in rate, and hence negligible in our design.

Packet loss will be handled by two mechanisms: Packet-level error correcting codes, and lost data retransmission. The ultimate fallback when all else fails will be retransmission of lost data. When data is retransmitted in QUIC as a response to a lost packet, the original packet is not retransmitted. Instead, the encapsulated data is placed in a new packet, and that new packet is sent.

For reduced latency of shorter streams, and for the tail-end of some (all?) streams, redundant information may be added by the protocol to facilitate error correction, and reduce the need for retransmission. For larger streams, where serialization latency is deemed to be the dominant factor (by the application that constructs and send the stream), the use of FEC redundancy may be reduced or eliminated for most packets of the stream.

CONGESTION AVOIDANCE

Although TCP uses congestion windows, we are currently planning on also experimenting with using a bandwidth estimator, and pacing packets evenly to not exceed the estimated available bandwidth. We also plan to monitor changes in transmission delay, and use that to detect that intermediate buffers are growing (available bandwidth was exceeded), or that intermediate buffers are empty (increase in pace rate does not impact latency, and one-way-latency is at a minimum). We expect this approach to give us a much smoother range of bandwidth transmissions rates, and a much faster tracking of available bandwidth (hopefully much faster than “TCP slow-start”). Similar technology is already in use in WebRTC.

Packet loss will be presumed to be a sign of congestion along the path of some packets in the connection, and will be handled analogously to the way TCP currently handles it. An explicit negative acknowledgement (NACK) will be transmitted to the source of the congested connection when a receiver decides that a packet loss has taken place.

In response to a NACK, a transmitter should alter its rate of transmission, such as by reducing the congestion window, or adding additional wait times between paced transmissions of UDP packets. The adjustment made to the sender’s transmission rate should be similar in spirit to what is performed by TCP. The response should be roughly equivalent, in overall impact, to what would take place in TCP if all multiplexed streams were separate TCP connections, and one of those hypothetical TCP streams received notification of a packet loss (congestion). For instance, we’d expect the sum of the congestion windows of those hypothetical TCP connections to change almost similarly to the way our single QUIC congestion window changes. The goal of this similarity is to be “TCP Friendly,” and neither dominate flows (impeding TCP activity), nor relinquishing control (allowing TCP to dominate utilization).

Using algorithms similarly friendly to TCP, bandwidth utilization may increase just as congestion control windows would be expanded when data acknowledgments are received.

ATTACK MITIGATION FOR OPTIMISTIC ACK ATTACKS

One known attack on TCP that could potentially be worse for QUIC is the Optimistic ACK Attack. In that attack, a malformed or potentially malicious client consistently acknowledges packets, even including ones that were not truly received. In TCP, if no packets are ever (reported) lost, then the bandwidth used by the sender (server?) can grow without bounds. The result of this attack is that a server may unwittingly flood an ISP, creating a DOS attack when a single rogue client misleads the server. With QUIC, our dependence on a receiver to evaluate the potential available bandwidth could cause a server to even more quickly DOS an ISP.

The defense against this attack is rather straightforward: The client must prove that it has received packets, and is not encountering packet loss, before a server should commit to high level bandwidth transmission. So long as the client is provably receiving all the packets, there is no reason to throttle the service. Once packets are (honestly) reported lost, the QUIC server will throttle the packet send rate (or outstanding packet window).

To support the probabilistic proof of receipt, each packet will include approximately one bit of entropy (one unpredictable bit, which is present in the encrypted payload). It is not sufficient to rely on the actual data in the payload to supply entropy, as that data may be predictable to an attacker (e.g.., the payload may have been downloaded before!). Similarly, the MAC and encryption might not be any more random than the data, given that those quantities may be deterministically constructed from the data. The one element of the packet that is at least slightly unpredictable is the time of transmission of the packet (which is actually included in the MAC region), but an explicitly random bit in each packet simplified the validation of the proof.

The actual proof of receipt will accompany each QUIC ACK, and it will take the form of a commutative checksum (i.e., a checksum that is not dependent on the order in which the checksum is created). The checksum will be 8 bits in length, and will include the entropy from all the packets that the receiver asserts it has received. If a server ever detects a mistaken checksum, it will presume that the cause is a malformed receiver, and will terminate the connection.

The one remaining issue with this mitigation is that the ACKs must include an explicit list of all packets that have ever been lost (and hence are not included in the checksum). Given that QUIC allows a sender to decide not to retransmit a lost packet, such an un-received list may grow over time without bound. To prevent this unbounded growth, when a sender provides an assertion that it will no longer retransmit packet numbers prior to K, it also provides the checksum that it expected to be associated with receipt of all packets up to and including packet K. As a result, a receiver will only need to list the packets after K, that

it is NACKing, along with the checksum it has accumulated (tallying the entropy in packets after the K'th packet atop the checksum provided for the K'th packet and earlier)..

If a receiver attempts to perpetrate this attack, it will be tested with each and every ACK that it sends. It is extremely improbable that a malicious receiver will be able to repeatedly guess the checksums it needs to provide valid acknowledgements.

PLAUSIBLE ERROR CORRECTING PATTERNS

Unlike traditional error correction on a continuous stream, error correction in the face of packet loss should make use of the fact that a whole packet is typically lost, or arrives intact. We presume that a packet level integrity check ensures packets are internally intact. This loss of "all or nothing" can exploit the same elements that RAID storage uses to change simple "error detection" schemes into "error correcting" schemes.

An example of a simple block-level error correcting scheme is to send N packet payloads, and then send the parity (bitwise sum) of all the payloads. That scheme can be shown to support 1 packet loss error ***correcting***. Although parity is traditionally only capable of 1 bit error ***detection*** (in this case one packet loss), the fact that the lost bits (if any) are in a known location, allows the lost packet to be recovered (assuming no more than 1 packet was lost). Such a simple parity scheme has the advantage of being scalable in terms of redundancy, as a small value of N implies high redundancy (additional bandwidth used), while a large value of N implies reduced redundancy and arbitrarily small excess bandwidth. An additional advantage of such a simple scheme is that the decision on the value of N may be made late, meaning that the sender may decide to use a smaller value of N (and send a parity packet) when there is no more data to send, or when the tail of a stream in encountered.

[One caution with using such error correction with large values of N is that the latency impact of a packet loss, and its associated error recovery, may grow in proportion to N. As a result, when bounded latency and/or jitter is required, the size of N should be constrained by roughly RTT * unidirectional_utilized_bandwidth/avg_packet_size. In that equation, "utilized bandwidth" is the data rate that is expected to be in use, and may possibly be less than the maximum available bandwidth. Essentially, if it takes the receiver more than 1 RTT to see the parity bits for the first packet in an N packet block, there is no latency savings (over a retransmission request).]

The above example is analogous to RAID 3 through RAID 5 configurations. Using a parity system analogous to RAID 6 based on Reed-Solomon codes, where two parity packets are sent following N data packets, it should be possible to correct for up to 2 packets lost in

the N+2 packets transmitted (without requiring a packet retransmission).

Experiments that were performed on Chrome sending UDP packets suggest that losses, when packets are paced, are decorrelated enough that the above simple XOR FEC strategy may be valuable for groups of packets. Unfortunately, there was some loss correlation, and attempts to recover from larger burst losses (2 packets+) provided diminishing returns for a large overhead. As a result, we will focus on XOR based FEC in QUIC.

When error correction is used, the plaintext payloads are gathered, a corresponding error correction payload is calculated, and an error correction packet conveys the encrypted payload. As a result, it should ***not*** be apparent to an eavesdropper when a packet containing error correction is transmitted.

PACING TO REDUCE PACKET LOSS

Experiments with Chrome have shown that packet loss can often be reduced by pacing packets. We believe the pacing reduces fluctuations in the packet flow, and thereby reduces congestion based losses (i.e., packet drop in routers due to overflows). We should try to use pacing to further decorrelate our packet loss statistics, increasing the chance that our packets will be interspersed from external flows.

The current plan for congestion avoidance is based on bandwidth estimation plus pacing, and hence pacing-reduced loss may fall out of the design. We may need some better hooks deep into the operating system to better facilitate accurate pacing in OS level buffering.

Even with the baseline TCP like congestion avoidance algorithm, we will attempt to use pacing to further reduce packet loss. For example, a TCP like algorithm maintains a congestion window size (count of packets that may be in flight), and a smoothed estimate of RTT. Dividing those quantities provides the equivalent pacing rate. We will attempt to not exceed that pacing rate, especially as we transition from quiescent to active transmissions.

RETRANSMISSION RECOVERY FROM PACKET LOSS

In cases where packet losses have exceeded the error-recovery limits of the protocol, a request for retransmission may be implicitly or explicitly produced. The techniques for instigating retransmission will be modeled after the TCP protocol. In keeping with that system, acknowledgements of data that is received will be periodically sent, and timeouts may be used to spontaneously instigate a retransmission when no acknowledgement is

received. Acknowledgments will also serve a key role in delivering congestion related bandwidth information, analogously to TCP's ACK's impact on its congestion windows.

PROACTIVE SPECULATIVE RETRANSMISSION

Certain packets, such as initial cryptographic negotiation packet(s), are critical to the latency of connection establishment, and all streams would block if any such packet were lost. In order to defend against such inopportune packet loss, these packets' contents may be retransmitted before there is any definitive evidence of such a loss. Any resulting duplicate arrival will be treated similarly to a duplicate transmission induced by a time out.

The advantage of proactive speculative retransmission rather than FEC groups (for example: having an FEC group with one data packet and one FEC/redundancy packet) is that FEC groups in QUIC will be transmitted sequentially, while speculative retransmission may easily be out of order, further reducing the chance that any correlated packet loss will impact more than one of a set of redundant packets.

As an example, a first (connection establishment) UDP packet in QUIC might contain proposed cryptographic credentials, while a second packet might contain (encrypted) requests for content. If either of those UDP packets were lost, then the connection would not be established, or at least the content would not be requested in a timely fashion. In this example, QUIC may wait a short period of time (e.g., 20ms? 0.25 x expected RTT?) and the retransmit those two connection starting packets. With such a gap between the initial transmission and retransmission of the crypto packet, it is especially unlikely that both will be lost.

BUFFER BLOAT

Buffer bloat has proved to be a consistently difficult problem, not just for TCP, but also for higher level protocols such as SPDY. We need to attempt to incorporate controls that will reduce the potential for being a large contributor to this problem.

Buffer bloat may be especially problematic to streams requiring timely delivery. Applications with timely delivery requirements, such as real-time audio or video, can often reduce bandwidth *if* they are made aware that "excessive" packet transmission will cause untimely delivery. We need to provide as much visibility to the application as possible, and work to have maximally just-in-time placement of packets into our local send queue, as well as just-in-time acceptance of packets from our application(s). This need will drive several of the API elements surrounding connection status, and notifications.

[HARD PROBLEM: We need to nail down a plan.] There is a fundamental problem that

concurrent TCP connections, and flows traversing routers or links shared by our connection, may drive buffers towards their maximum size. If we act kindly towards bloated buffers, and reduce traffic when or if we do detect bloat, then we would progressively be driven to having a smaller percentage of any bloated buffer containing our traffic. That reduction would in turn result in smaller percentage of the egress bandwidth at said buffer being dedicated to our flow, until we were starved. If we transmit as aggressively as TCP, then we would share "fairly" in the egress bandwidth, by proportionally populating the bloated queue, but we would also be a contributor (if not the solo creator) of buffer bloat. We need to decide on a philosophy for walking this line. [Perchance we can take a forward looking view, anticipating better handling by routers, being more conservative than TCP, but not so conservative that in today's landscape we would neuter the value of this protocol. This is IMO a very hard problem.]

LOCAL BUFFER CONTROL

One place where buffering is potentially controllable, and observable, is on a machine associated with a sender endpoint. A sender machine typically has the potential for extremely high local bandwidth, as measured with the ability to fill an output buffer, and this can be markedly higher than the bandwidth on a link-level egress, such as wired Ethernet, or Wi-Fi. We need to try to make this protocol aware of the local current queue size (to the extent the hardware and OS will allow), and work to minimize the actual size, subject to avoiding starvation of the local egress link for lack of efficiently buffered data.

STREAM-BASED FLOW CONTROL

Flow control provides a basic ability for receive-end status, such as excessive use of (buffer) memory, to induce a request that the send-end diminish (or in the extreme, cease) transmission. With a multiplexed protocol, we have the potential that a consumer for one stream at the receive-end to be slow, or not consuming data, while other source-sink pairs may be proceeding "normally." The fact that we have out-of-order delivery ensures that we won't block the other streams, but we may face receive-side (buffer/memory) resource exhaustion due to unconsumed stream packets.

[We may model a resolution after work being done in SPDY to handle such stream-based controls, but we may have additional controls that can be accessible. For instance, in addition to requesting a reduction in a specific stream's send-rate, we may also discard (and not ACK). Such an approach may not only provide back-pressure to the send-side, it may also partially alleviate overgrown resource usage pressure at the receive end. This can be contrasted with TCP, where send buffers are saved until received at the OS level, rather than until (acceptably) received at much higher levels in the protocol.]

IDLE ENTRY

At some points, the overall connection will be idle, as there will be no data in any streams pending transmission. In TCP, trailing packets that are lost as we enter an idle state (a.k.a., tail drop) may induce a timer-based retransmission. Such retransmissions have the potential for adding significant latency, and error correcting redundancy should be used to reduce the probability of such retransmissions.

If we are already using an error-correcting pattern as we approach an idle state, then nothing needs to be done, as a trailing error-correcting packet can be added to the connection as soon as we actually go idle (run out of data), or shortly thereafter. If we are transmitting a stream in a mode that does not have error correction, then we may need to start using error correction as we approach the idle state.

Conceptually, we should add error-correcting redundancy as we approach an idle state with any stream, even if the associated stream does not generally use error-recovery. We should optimally begin this phase as soon as we anticipate that we may enter an idle state within less than one RTT.

IDLE DEPARTURE

Periods of idle (inactivity) on a stream may induce either end of the connection to discard some or all state, or induce some middlebox (e.g., NAT router) to discard state. We need to account for each such loss of state, and provide efficient (low latency) methods for continuation.

NAT TABLE RESET

NAT boxes may have their entries time out, or otherwise expire, causing a breakdown of the underlying UDP transport. Such expirations are common today after 30-120 seconds of inactivity. After such a NAT expiration, a server may be completely unable to initiate any data transmission to a client (until the client re-establishes a NAT forwarding port).

The section on NAT Table Keep-Alives describes a process for adaptively handling this issue.

CONTINUATION WITH FULL CONNECTION STATE

A second example of idle departure may appear when a client seeks to continue transmission of data on a previously established connection. With TCP, this class of continuation is allowable until a RST has been sent, and many servers currently reset TCP connections after about 5 minutes of inactivity. With QUIC, given the fact that a 0-RTT re-connection can generally be made using credentials acquired during a previous connection, we are initially planning on only keeping a connection alive (with no streams open) for about 30 seconds. This should also allow us to more commonly avoid NAT rebinding complexities.

Given concerns about battery life, and uselessly waking up a receiver to say “we are done,” we negotiate over the connection how long the connection state is *expected* to be maintained at both ends, after the connection has gone idle. This negotiation will take place during regular communications on the connection, but will take effect automatically when the channel goes idle. Resumption of traffic within this time window will ensure that the cryptographic elements of the connection, and any other session state (such as a compression context), are still valid/available.

CRYPTOGRAPHIC ELEMENTS

Initial negotiation of a cryptographically secure and authenticated channel, at a site that has never been visited previously by a client, will be significantly modeled after TLS, and a traditional TLS HELLO message exchange. As such, the absolutely first interactions will require one full RTT prior to any data transmission by the client. After a first introduction, including negotiation of cryptographic parameters, and exchanges of certificate chains etc., a client may record the results of such activities, and use them for performing a more rapid future connection.

RESERVED CRYPTOGRAPHIC COMMUNICATIONS STREAM

All cryptographic exchanges tend to use in-order delivery of a set of messages, where that fact is critical to the analysis and correction proof of the protocol. As a result, we will reserve a stream, with Stream ID = 1, the CryptoStream, as the cryptographic meta-information communications channel. It is given the stream ID 1 because it is client instigated, and hence must be odd, and is started in the initial connection message. Using a reserved stream ID will facilitate re-use code for QUIC streams to support our cryptographic exchanges, which are strongly modeled after TLS.

CRYPTOGRAPHIC HEAD-OF-LINE BLOCKING

Two methods will commonly be employed to prevent this cryptographic stream from encountering a packet loss, which in turn causes head-of-line blocking on one or more concurrent streams. For example, if a part of a key negotiation is lost, we wish to reduce the probability of having a retransmission delaying decryption of later packets.

The first method of reducing the probability of head-of-line blocking is speculative/redundant transmission of critical packets. For example, the initial cryptographic connection establishment may be routinely transmitted more than once. In addition, the redundant retransmission(s) may be delayed slightly from the initial transmissions to significantly reduce the potential for correlated packet loss of all (both?) copies.

The second method focuses specifically on points in time where the cipher specification changes (e.g., an encryption key changes), similarly to where a TLS Change Cipher message is employed. Rather than using an explicit message (serially delivered relative to key use), a new encryption/decryption key is brought into use asynchronously. The authentication by the new key signals the transition to a new decryption key. A sender will usually wait until it has received acknowledgement that data in the CryptoStream updating the receiver on encryption key has been received before proceeding to use a new key. When this method is employed, head-of-line blocking (for lack of an encryption key) is impossible, and a receiver may discard an old key after all packets with sequence numbers prior to a use of the new key have been processed.

ENCRYPTION AND AUTHENTICATION

We will use AEAD (Authenticated Encryption and Associated Data) to protect the bulk of each UDP packet. The negotiation of keys is more completely described in a separate QUIC Crypto document.

We will avoid serialized decoding dependency in QUIC (which would damage our ability to provide out-of-order delivery in the face of packet loss) by deriving an IV for each packet from the UDP packet sequence number. In QUIC, we deliberately partition data so that each independently decryptable section (using an IV) falls entirely within a single UDP packet, and there is only one such section in each UDP packets.

SESSION KEY UPGRADE

There are several reasons for changing a session encryption key, during the lifetime of a

connection. The first change is a transmission from a NULL encryption (authentication only, with a known key). Initial packets sent by a client containing the cryptographic Client HELLO must be transmitted without encryption, but data in subsequent packets will potentially be encrypted per negotiations. The second example centers on speculative connection, which may depend on a session key for which perfect forward secrecy is not ensured. After speculative connection establishment, we will change to a session key for which perfect forward secrecy is guaranteed.

Rather than sending the moral equivalent to a TLS Change Cipher Suite message, and risking confusion and head-of-line-blocking if that message is lost, we rely on trial decryptions during transitions. When a receiver becomes aware of the potential to use a new key, it can try both the new and old key until the new key succeeds. At that point, it can begin using only the new key for larger (future) packet sequence numbers. If an endpoint receives an acknowledgement that proves that the distant endpoint is also aware of a new key, it may also proceed to only accept packets (with larger sequence numbers) decrypted with the new key.

As a result, the cost of a transition will be minimal (the time when only one endpoint knows about a new key is roughly bounded by an RTT), and a third party should not be able to detect a transition when examining a packet.

PROTOCOL DETAILS: SPECIFICATION RATIONALE

DEPLOYMENT ISSUES

Initial experiments with UDP connectivity from browsers around the world suggest that roughly 90-95% of users will have adequate UDP connectivity for successful QUIC connections. We conjecture that the 5%+ user connectivity block is predominantly caused by LAN firewalls, probably in enterprise settings. In order to compensate for this, all QUIC connection establishments will routinely be raced against more common TCP (and often TLS) connection. Once the QUIC connection is active, requests will continue to traverse the connection without races.

ALTERNATE PROTOCOL HEADER

There is currently no supported scheme, such as HTTP or HTTPS to indicate that a resource should be acquired over QUIC, and so we will depend on Alternate Protocol Headers in resources to convey support of QUIC. To facilitate this, clients and servers are encouraged to employ Server Advertisement of QUIC through the HTTP Alternate-Protocol

header

When a server receives a non-QUIC request which could have been served via QUIC, it should append an Alternate-Protocol header into the response stream, analogously as was proposed for SPDY. We currently plan to routinely support QUIC on either UDP port 80 or UDP port 443 to convey traffic that would otherwise have been supplied via HTTP or HTTPS respectively. In both cases, we will be encrypting the traffic (to avoid accidental damage by middle boxes), but we may (TBD) have reduced certificate validation on UDP port 80.

To specify QUIC as an alternate protocol available on port 80, use:
Alternate-Protocol: 80:quic

If the resource was requested over HTTPS, then we will employ equally strong certificate chain validation on the QUIC connection as is required by TLS.

INITIAL (CONNECTION ESTABLISHMENT) PACKET DEFAULTS

All packets, including the first packets which negotiate a connection, will utilize AEAD encryption. The first packet(s) will used a default null-encryption key, and the AEAD will then serve only to preclude accidental tampering (act as a high quality checksum).

Cryptographic negotiation will take place on Stream 1, created by the client.

PROTOCOL OVERVIEW OF ELEMENTS

At the highest level, each connection (related set of UDP packets) is associated with a GUID (a Globally Unique ID, selected at random from a large enough space that there is negligible chance of a collision). Packets are numbered sequentially, including all data, control, FEC (Forward Error Correction), and acknowledgment packets. FEC covers various ranges of data, and is appended lazily, so the protected data does not indicate the quality or immediacy of the FEC data that follows.

Connections can be resumed very quickly, and we assume that when there is a risk of a connection time-out (other side of connection dropped state; or middle box dropped NAT state), a transmitter can prefix a transmission with startup packets (that establish a connection, typically with no additional round trips).

All packets are authenticated and encrypted, when cryptographic contexts are available (i.e., except some initial hello related packets on connection). A connection reset is also authenticated, and hence cannot be directly instigated by a third-party intermediary (although if enough packets are blocked by an intermediary, the connection may eventually be abandoned).

Although we need to acknowledge receipt and loss of UDP packets (to in part, handle congestion), a transmitter does not always need, or choose, to retransmit packets. This is most apparent when an FEC packet is lost, as it is always preferable (as an ultimate fallback) to retransmit protected data (if it was lost!). The fact that a transmitter has the option of not retransmitting a packet (and a recipient has generally no idea what the lost packet contained), automatically provides low level support for a higher level protocol that provides stream data that does not need to be re-transmitted (example: can't be re-transmitted in a timely fashion, and be usable). Since a transmitter may choose to not retransmit, it must provide acknowledgment-like packets indicating the intent to not retransmit (which is much less data than a retransmission).

FRAMING

This section is intended to describe information about bytes-on-the-wire, and some sequence of transmissions. Bit specific details will be documented in a Wire Specification, but this document will serve to explain the justification for the fields, and their potential use.

The discussion will start the more detailed description of the protocol by looking first at the steady-state transmission of data. This will establish a lot of structures which can then be re-used when discussing distinct control related packets, such as acknowledgement related packets, FEC packets, and connection establishment packets.

QUIC PACKET FRAMING OVERVIEW

The basic unit of transmission will be a standard UDP packet (a.k.a., a packet). Care will be taken to ensure that all data transmissions will be broken into blocks that fit cleanly into a single packet.

All QUIC packets consist of a header section, and a payload section. A data packet's payload contains a sequence of frames. An FEC packet's payload contains redundant information. The payload in each packet is an AEAD encrypted block. The associated data (that is also authenticated) includes the entire header.

The header for each packet consists of:

- Public Flags - 1 byte, detailing the layout of the rest of the header
- GUID
- QUIC Version
- Packet Sequence Number

Header: Public Flags

The header is potentially large, and several of the fields are at times unnecessarily large, or even completely unnecessary. The Public Flags encode the sizes of each of the other header fields, and allow for more compact representations. Simply put, the Public Flags provide per-packet specs for the sizes of the other fields in a given connection, in a given packet.

Header: GUID

The GUID is specified to be 64 bits in length so that clients can randomly select a GUID, and contact a server at a fixed port, and yet have a low probability of colliding with other connections. By the time a server hosts about 2^32 concurrent connections, there will be about a 50% chance that *one* attempted connection will collide with an existing connection. At that point, there will be 1 connection in 2^32 that arrived at a conflict, and will get “unusually poor” connection performance (it will be probably get a time-out, as its packet will not authenticate, and will be discarded). The user that temporarily gets such a problem will, in a good implementation, automatically fallback to making a connection via TCP.

The GUID is however completely redundant for a client that has created a dedicated ephemeral port to contact a server. The *only* packets that the client will receive on that port will be part of the singular QUIC outgoing connection. As a result, a client may request that a server not bother to include a GUID in each and every packet. Once a server receives such a request (such as during connection negotiation?), the server can use the Public Flags to indicate that the GUID is omitted (has length 0 in the header).

For some servers and services, the number of parallel connection connections is heavily restricted. For example, a server might negotiate with a client to continue the connection at a specific alternate IP and port. In that case, the server could also indicate to the client that a smaller GUID may be acceptable.

Header: QUIC Version

We know the protocol is evolving quickly, and needs to evolve. This field is present in the first packet to ensure that the server can understand the same version as the client will provide. Once the connection is established, this is really redundant, and the Public Flags will then indicate that this field is omitted (has length 0). If a server needs to distinguish the version, it will remember that the connection, defined by the GUID, has a distinctive version.

Header: Packet Sequence Number

In addition to sequencing packets, watching for duplicates, and communicating what packets are missing, this number is a critical part of the encryption. This number forms the basis of the IV used to decrypt each packet. As a result, it must conceptually be large, as it must never repeat during the lifetime of the connection. That need forced this sequence number to be conceptually large, around 2^64 (more packets than any connection would dream of sending), but we usually don't need to provide all 8 bytes in each packet!

At any given time, where will only be a finite (small) number of packet sequence numbers that have not been acknowledged. This restriction is a natural consequence of the fact that a sender must buffer internal data for those pending packets, and the sender's memory is very finite. In addition, we have chosen to not "retransmit" packets that are declared lost, but rather "rebundle" their contents in later packets. As a result, a receiver will often communicate that it has not received a packet, and then the sender will notify the receiver to "stop waiting" for the packet, and hence the window of unacknowledged packets will always be nicely bounded, and the ranges of uncertainty (max and min possible being discussed) can be known by the sender. Based on that restriction, a sender can significantly reduce the number of bytes needed to express the packet sequence number (using the Public Flags).

As an example, suppose that packets are being transmitted with a TCP like congestion control, and the current congestion window is 20 packets. Even if a packet is lost, within 1 RTT, or about 20 additional packets, a receiver will be informed that a lost packet is no longer pending. As a result, a sender can get away with sending only the low-order byte (8 bits) of the 64 bit packet sequence number on the wire. A receiver can easily decide based on those bits what the upper 56 bits must be, and can then use that to decrypt the packet. Note that *IF* a very old packet ever arrived at a receiver, and the old packet sequence number was then misinterpreted, then the AEAD authentication would fail, and the old (and agreeable discarded) packet would indeed be ignored.

not applicable in curvecp

Payload Framing Overview

After a header in a packet, there is always a payload block of ciphertext authenticated by an AEAD algorithm. That block consists conceptually of some number of redundant authentication bits, concatenated to a string of bytes the size of the plaintext of the payload. We currently estimate the need for about 64 bits of authentication, but that will vary over time with the state of the art, and be negotiated by the cryptographic exchanges.

After decrypting, we will have a plaintext payload block that will consist of:

- Private Flags - 1 byte
- FEC Group number
- Series of self-identifying Frames

Payload: Private Flags

The Private Flags, currently 1 byte, are denoted “private” because they are shrouded by encryption, and not visible to an eavesdropper. As with the Public Flags, one value encoded in the flag is the size of the FEC group number, and implicitly the offset in the payload to the start of the frames.

The Private flags also have 2 other individual bits. The first bit is a random entropy bit, and the second bit is used to identify the last packet in an FEC group. The last packet in an FEC group is the redundant FEC packet (containing the XOR of the plaintext payloads in the group).

Private Flags: Entropy Bit

The Entropy bit is randomly selected by a sender, and placed into each packet. The information is used to combat any potential Optimistic Ack Attacks. When a receiver sends acknowledgements for a set of packets, it is required to prove it received the set it asserts by providing the hash of the Entropy Bits it claims to have seen.

One issue with Entropy Bits is that when a packet is lost, its entropy bit is not seen. To handle that, and not require a receiver to create and endless list of loss packets (exceptions from the hash), the sender regularly provides an update that indicates what the hash *should* have been, up to a specific packet. It only provides this update after it has gotten truthful assertions about missing packets prior to that point.

As an example, suppose packets 1, 2, 3, and 4 are sent. Assume packet 3 is lost. A

receiver will indicate (in an ACK frame) that it received all packets up to 4, except for packet 3. That ACK frame would also provides a hash of the entropy bits it packets 1, 2, and 4. The sender can confirm that hash is correct, and declare packet 3 missing. The sender can then transmit (within an ACK frame) that it is no longer concerned with packets 4 or older, and it can provide the hash of all entropy bits up through packet 4. As a result, eventually the receiver will know the hash for packets up through packet 4, and can provide additional incremental hash results as it receives further packets.

Private Flags: FEC Final Bit

This bit is used to mark the last packet is an FEC group. Our current FEC approach has exactly one FEC packet at the tail end of a series of packets protected by FEC. This bit indicates that the entire payload should be handled separately, and viewed as the XOR sum of the other payloads in this group.

Note that when this bit is set, the packet must be part of some FEC group, and hence the FEC group number must always be present as well.

Payload: FEC Group Number

Given that we routinely see 1%+ packet loss on the internet, it is extremely unlikely that over 100 packets can be received without a loss, and certainly not 200. For this reason we decided to limit FEC groups to be no larger than 255 packets. Our experiments suggest that when FEC is used, it may be most beneficial in the range of a 10-20 packet data sequence, protected by a redundant FEC packet. With 20 packets, there is a bandwidth cost of about 5%, and there is a very visible tradeoff against latency. We also noted that with TCP, it is most common to see congestion windows well under 50 packets, and so the use of an FEC for larger groups would not be beneficial in reducing latency, when compared with retransmission.

As a result of the above analysis, we currently only support simple XOR FEC of a consecutive set of packets, and one byte is sufficient to identify a group of FEC protected packets. If and when a sender decides to protect a set of packets, it uses the Private Flag to indicate that there is indeed an embedded FEC Group Number byte. Each participant in an FEC group has an offset FEC Group Number that can identify the first packet in the group (i.e., it is an amount to subtract from the current packet sequence number). This approach allows for a lazy decision as to when to end the FEC group (i.e., the exact number of packets need not be known, and the final FEC packet can be added at any time, such as when there is no more data to send!).

Payload: Self Identifying Frames

The bulk of each QUIC packet is expected to be a concatenated list of data called Frames. Each frame has leading bytes that identify the frame, as well as information about the format of that frame and its contents. For example, there are frames that carry acknowledgement information, and also frames that carry individual stream data. There are also a variety of miscellaneous frames.

Packets are generally packed until they are full with data contained in one or more frames, and then sent. Frames are in a sense the workhorse of the protocol, and are discussed in more details in the next section.

Frames Within the Payload

There are a variety of Frames currently defined to be sent within the payload. This section will provide some motivations for their use and structure.

All frames start with a Frame Type byte, but we expect to pack additional flags specific to each type into that byte. As a result, it will actually be the first few bits of a frame-type byte that identify the kind of Frame, and the other bits will be used as flags to encode formatting within that Frame. Initial implementation versions of QUIC may use very fixed size fields for many of the Frame Types, but that should change as we further optimize and compact the encodings.

STREAM_FRAME

Each QUIC connection can multiplex a collection of streams, and each Stream Frame conveys application data for a single stream. The frame's header conceptually must present the stream number that the Frame is a part of, plus the starting offset within the stream for the contained data (just as TCP provides byte offsets for segments). A Stream Frame is also used to implicitly open or create a stream, and the frame currently contains a flag bit that indicates that it is the last portion of said stream.

In order to increase efficiency, which may be critical when a large file transfer takes place on a single stream, we expect to employ variable length fields in the header for this frame. For example, even though we expect to support very large file transfers, it will be rare that a stream will convey data that uses many of the current 64 bits of maximum offset! Similarly,

when large streams are being transported, it will be very common for a single data frame to fill the entire remaining space in a payload, and hence its size (within the payload) can be implicit rather than explicit.

As with SPDY, we expect to use compression of stream headers for HTTP requests. This should be negotiated during the connection establishment and the cryptographic handshakes. One cost of this approach is that the stream headers must be sequentially decoded, and hence a new stream cannot be processed until the header of the previous stream is decoded. We expect that in many cases this compression of HTTP headers will be so significant, that a multitude of HTTP requests will fit nicely into a single QUIC packet, and this head of line blocking will be insignificant. We also have expectations that HTTP/2.0 will settle on an even further improved compression algorithm, with better control of serialized decoding, and that will then be employed in QUIC.

ACK_FRAME

ACKs in separate per-stream frames

An ACK Frame is used to coordinate packet loss recovery, and is kindred, but not identical to an ACK packet in TCP. An ACK in QUIC is always cumulative, in that new ACKs contain enough information that any prior ACK should be discarded. As a result, if a packet containing an ACK Frame is lost, it is not necessary to retransmit the enclosed ACK Frame.

versus always selective ACKs in SST

The first and most clear conceptual contents that an ACK Frame contains is a NACK list of packets that a receiver has decided are missing. As a result, an ACK Frame has a count of the number of missing packets, plus the list of packets. Just as with the Packet Sequence Number wire presentation in the QUIC Header, the missing packets can be very compactly represented. Even with a compact representation, there is a limit as to how many NACKs can be enumerated within a single packet (several hundred). As a result, in the super improbable case where there are too many NACKs to fit in a packet, only the eldest (numerically lowest) missing packets are cited. This improbable case, caused by 1% packet loss, could limit the effective in-flight window to a few 10,000 packets (not a big limitation... but even the edge cases must work). We refer to such a case as a “truncated ACK Frame.”

A second conceptual element present in the ACK Frame is usually the largest packet sequence number that has been received, and we refer to that as the “largest observed” packet. That structure makes an ACK Frame similar to a TCP Selective ACK, which lists the largest received offset, plus a small itemized list (as many as a pair) of NACKs, if any.

One complication with QUIC is that an ACK Frame can be truncated as noted. In the case where the ACK Frame is truncated, it is possible that the receiver was missing several packets at this truncation boundary. To properly handle that odd case, the “largest observed” packet, present in the ACK packet, may *also* be among the NACKed packet list!

To make the above issue clearer, consider a receiver that was sent packets 1 through 1000, but only received packets 1 and 1000 (i.e., it was missing packets 2 through 999). In that example, we’ll assume that there was only room for 200 NACKs in an ACK Frame. As a result, an ACK Frame listed (explicitly) NACKs for the 200 packets 2 through 201. Oddly, the “largest received” packet was packet 1000, but that cannot be stated in the ACK Frame, as it would look like packets 202-999 were received! In this truncated ACK Frame case, the “largest observed” packet would have been listed as packet 201. The above situation would rapidly resolve itself as the sender would respond to such NACK list with a request to not consider NACKing packets 201 or older. Bottom line: even in this edge case, forward progress will be made ;-). just use the curvecp-style range ACK

A second component of an ACK Frame conveys information in the reverse of the common direction, as it carries acknowledgement information from a sender to a receiver. QUIC does not retransmit packets, but only rebundles their contents in a later packet if they are still useful (i.e., it will tend to retransmit Stream Frames in lost packets, but not retransmit ACK Frames from lost packets). As a result, when a sender learns via a NACK of a lost packet, it needs to notify the recipient to “stop NACKing” that packet. To support this, an ACK Frame also has a field to identify a “least unACKed” packet. This packet sequence number identifies a point for which the sender does not want any older NACKs.

The last two elements provided in an ACK Frame are related to the entropy hashes used to preclude an optimistic ACK attack. As mentioned earlier, an entropy bit is provided in each packet in the payloads Private Flags. When a receiver sends an ACK Frame potentially specifying packets it is aware of, and packets it is missing, it also sends a Received Entropy hash for the packets it is aware of. Similarly, when a sender updates the Least Unacked packet number, it also provides the hash that a receiver should use as the accumulated entropy up to that Least Unacked point.

In general, a packet containing only an ACK Frame will not be ACKed (which would otherwise lead to an endless volley of ACKs!). A missing packet that follows a packet containing only an ACK Frame will of course be NACKed, as the receiver will have no idea what the lost packet contained.

When a sender receives an ACK Frame containing NACKs (especially new NACKs, that it has not previously heard about), it may provide a responsive ACK Frame that asks the recipient to “stop NACKing” said packet(s).

As with TCP, when a sufficient length of time (relative to RTT) has passed without receipt of an acknowledgement, a sender may retransmit the contents of a packet (i.e., act as though an explicit NACK was provided). Such retransmissions will proceed to use an exponential backoff, but just as with TCP, assume that when a packet (contents) are being retransmitted, they are not restricted by a congestion window (i.e., the presumed loss allows for the retransmission).

CONGESTION_CONTROL_FRAME

The Congestion Control Frame is used to convey information relating to a specific congestion control algorithm. At the start of the connection, the negotiation of cryptographic credentials also includes the agreeable negotiation on a congestion control algorithm. As a baseline, all implementations will support a TCP-like algorithm, but we expect other algorithms to be used, with the need for correspondingly different data.

For example, in a TCP-like algorithm, a Congestion Control Frame might contain information about when a packet (that is being ACKed) was received, or similarly, how long after being received was the acknowledgement sent. Such feedback statistics may then be helpful in calculating RTT, much as TCP uses the arrival of a TCP ACK to estimate an RTT (where presumes that there was negligible latency between receipt of a packet, and a corresponding ACK). The current TCP-like implementation of a congestion feedback frame includes the accumulated count of lost packets.

In the case of an algorithm that is focused on measuring sojourn latency, a congestion feedback frame might provide relative arrival time between packets (to allow the sender to deduce the packet spread), or statistics about receive times. For example, it might include periodic updates on what the shortest observed interarrival time for packets was. It might also include periodic statements of what the arrival time was for a specific recent packet, as measured by a receiver clock. In that case, the receiver clock is not synchronized with the sender’s clock, but the sender can detect a drift that is indicative of an increasing or decreasing intermediate packet queue. Such a statistic may be a much more precise estimate of the one way transit time (though its accuracy is impeded by the clock skew between endpoints).

The precise fields that will be present in a Congestion Control Frame will vary significantly based on the employed algorithm.

RST_STREAM_FRAME

The Reset Stream Frame is used to abnormally terminate a stream. It can for instance be used when a receiver wishes to prematurely stop the sender from sending additional data, or when the data source is no longer supplying data. To assist in debugging, a Reason Phrase is included in these Frames, as well as an Error Code.

CONNECTION_CLOSE_FRAME

The Connection Close Frame is used to aggressively and immediately abort a connection, closing all streams implicitly. To attempt to support such rapid teardown faster than is common in TCP, this Frame is also followed by an ACK Frame, to better communicate the state at the time of the teardown (i.e., what was the last packet sequence number received before the tear-down).

GOAWAY_STREAM_FRAME

The Go Away Stream Frame is a request for a graceful termination of a connection, where no new streams will be created, and only existing streams will persist (and hopefully finish shortly). This frame also includes a Reason Phrase, and an Error code.

CONNECTION RESET ALTERNATIVES

A QUIC connection can be reset by either side of the connection. Connection reset implies that the connection is permanently discarded, or torn down, and that no additional communication will be performed using packets associated with the connection's current GUID. Unlike TCP (including SSL over TCP), a QUIC connection cannot be torn down by injecting a single universal packet (e.g., a TCP RST packet). A connection will be (eventually?) abandoned and effectively reset when enough traffic has been lost, and no ACKs received, so a middlebox can eventually reset a connection, but a middlebox cannot reset the connection in an expedited manner as can be done with TCP.

There are two ways to communicate a QUIC connection reset. Both methods are authenticated (precluding third-party injection). The first method is via a standard packet that is authenticated identically to all other authenticated packets in a connection, by including a Go Away Frame, or a Connection Reset Frame (both described above).

The second form of reset uses a unidirectional pre-arranged reset nonce, which is a shared secret. We refer to this form as a Public Reset Packet because a third party can detect the presence of such a packet, although a third party cannot forge such a packet. Each reset nonce, for a given GUID, is optionally specified by each end of the connection

(to the remote end) early in the connection traffic. Once a reset nonce has been specified, the nonce may be used by the side of the connection that generated nonce to expeditiously reset the connection.

An expeditious reset may be needed if a packet is received with a GUID that is no longer associated with an active QUIC connection (and is not a startup packet). The unrecognizable GUID (and associated packet) could alternatively be ignored (discarded), but that would only cause a slow abandonment of the related connection by the sender.

For example, in a client-server connection, a server may select a reset nonce that it may (optionally) use in the future if it needs to expeditiously reset the connection. This nonce can be used after the session key for a given GUID has been discarded or lost by the server. Session keys may be discarded by a server shortly after a connection is torn down, or lost by a server during an unplanned server restart (reboot). The nonce may typically be constructed as a MAC of the GUID, where the MAC key is never revealed. Such a construction can allow (in this example) the server to re-construct the nonce for a given GUID, even after the connection, and associated cryptographic state, has been forgotten.

When a Public Reset Packet arrives, we can assume that the recipient still has copies of all state involved in the connection based on the GUID, including the pre-arranged reset-nonce. As a result, the recipient can validate a proof of possession relative to the packet that it transmitted (and was discarded), agreeably reset the connection, and proceed to construct a new connection. Note that the nonce is not transmitted in the clear, and instead a time-bound proof of possession of the nonce is transmitted.

MALICIOUS RETURN ADDRESS REWRITING

A malicious middlebox adversary could re-write the source IP or port, and cause a receiver to be “confused” about where to send response traffic. Note that neither the source IP nor Port are protected as associated data in the AEAD encryption, because NAT boxes must routinely rewrite these fields. Such rewrite malice by a middlebox could potentially force a receiver to at least consider responding toward the new source IP address. Potential malice will (at a minimum) complicate the details of the algorithm to adapt to mobile changes in source address and ports.

One way to avoid such a fallacious reset is to continue to direct packet traffic to the previous source IP, until a “sufficient” number of packets have been responded to from the new source IP. For example, we could temporarily transition to sending FEC groups of size one (meaning that the FEC packet provides 100% redundancy), and route all data

not an issue in curvecp due to public key signature in incoming packet

packets to the original source IP, and all FEC packets to the new source IP. This would minimize any interruption in the QUIC connection, and would allow authenticated ACKs to clarify when (and if) it is correct to switch completely to the new source IP.

Another attack that may be constructed by a client (or a middlebox) is to deliberately falsify the source IP address, creating a traffic amplification attack. In that scenario, a client may establish a connection to a server, demonstrate high bandwidth, request a large set of resources, and then the client/attacker could maliciously alter the underlying return address. In that example, a server could plausibly send a large quantity of data to the new source IP, unwittingly attacking (flooding) that source IP. The natural way to foil such an attack is to avoid extensively using a replacement source IP until at least one round trip of one packet has proceeded. We may be able to use the some of the same techniques that allow zero-RTT connection establishment, so as to support more rapid source IP utilization. For instance, we may be able to switch faster when we are under a low load; we may be able to check for any apparent DOS targeting of a small IP address range; we may restrict the amount of unACKed data sent to a new IP address.

TBD: Should a receiver be slow(er) to respond to a source IP or port change? Can we tolerate reduced latency during a transition, or would we be creating a partial DOS attack vector? Perhaps QUIC should wait for several packets with new return addresses before altering current responses, and potentially resending previous responses. This is a cat and mouse game, and the critical point is to force the malicious middlebox to expend significant effort, and to try to minimize the wasted effort and misrouted bandwidth of the victim. Is this effort futile?

The fact that middlebox malice is possible suggests that there may be limits on how well/quickly this protocol can adapt to mobile changes by a source in its address. If a source is aware that its address is changing, it may be possible to send an authenticated hint that a change is imminent, without being able to authenticate the details. Such a hint may be especially helpful, for example, as a mobile device switches to/from Wi-Fi, and hence can send assurances to a server that a transition is taking place.

CRYPTOGRAPHIC STARTUP OVERVIEW

There is a separate QUIC Crytpo document providing details, and justification of the precise cryptographic HELLO protocol, key exchange, etc. This section is only meant to conceptually outline the sequence of events that are involved in a connection establishment. The QUIC Crypto document is authoritative.

A typical 0-RTT Refresh Connection Establishment will be built around speculation by a client that a server still is using a previously known public key. Based on that speculation, a client may transmit a proposed session key, etc., that is encrypted with the predicted public key. The client can then immediately follow that Connection Establishment transmission with encrypted data, encrypted under that initial session key. This provides the fundamental, high probability, 0-RTT startup.

The above approach will allow at least initial encryption of data. In that state, the connection will be encrypted and authenticated, but will not exhibit perfect forward secrecy.

Under the protection of the initial session key, elements of the cryptographic HELLO exchange can then exchange cryptographic information sufficient to formulate an improved session key, which can provide perfect forward secrecy. After exchanging that information, each party can immediately begin to use the improved session key.

To reduce the likelihood of an unexpected loss of connection, each side of the connection should indicate how long it expects to maintain its session state, once the connection has become idle.

1-RTT Fallback

If the initial speculated public key in a connection establishment is not acceptable to a server, or the server wants additional validation of the requesting client's address, the server may Reject the connection request and perform a round-trip verification before proceeding. The round trip elements are expected to include the moral equivalent of a nonce (kindred to a TCP SYN-cookie), and to be the moral equivalent of a full HELLO exchange as performed in TLS. After the client responds to that server's hello, the client may retransmit any data requests (presumably discarded by a server decline), under the protection of the fully negotiated session key. In this example, the server may optionally discard all previous data packets, presumably sent under the protection of the speculated public key.

When a client is not yet familiar with a server, at least one RTT will be required in a connection establishment before traffic can be sent. In this case, the client cannot predict the likely server public key, and will send the moral equivalent of a TLS HELLO message. The server can then respond with either an addressing challenge, kindred to a SYN-cookie (if source verification is required and insufficient), or respond with the moral equivalent of the TLS HELLO protocol, including optionally an addressing challenge nonce. [The goal is

to minimize state and computing cost to the server, to the extent desired by the server.] The data in this extended HELLO sequence may contain a cert chain which will typically not fit in a single UDP packet, and will instead use a more extensive stream.

Either side may start transmitting normally as soon as the above HELLOs have been transmitted. In some cases, a significant amount of data may need to be transmitted in order to establish the identity of one or both parties. This information is transmitted in streams, with the usual congestion control and retransmission of streaming data.

2-RTT Worst Case Fallback

In the event that a server needs to respond with an extensive certificate chain, a server may decline to send a multi-packet payload to a client until the client can prove it is control of a specific source address. In that scenario, the HELLO exchange may take as many as 2 RTTs.

In this worst case, the first client HELLO would be rejected, but the server would supply a token to prove ownership of the apparent client address, as well a server certificate, without including a certificate chain (if it is too large to fit in a single packet).

The client would then use the source address token in a second HELLO request, but might be unwilling to trust the certificate until it receives the entire certificate chain! In that example, a server would then reject the second HELLO (for lack of any real proposal for a shared key), but would at least respond with a fully fleshed out certificate chain, requiring several packets to send.

Finally, with a trustworthy server certificate in hand, and a trustable source address token, a client may provide a client HELLO as suggested in the 0-RTT case, and the connection would proceed.

PROTOCOL GLOSSARY

Amplification Attack

An attack where an attacker sends packets to a server, and the server responds with more packets than are sent. This allows an attacker to leverage a server's potentially large bandwidth to greatly amplify the rate at which packets are directed towards a target, which then suffers a DOS attack, due to a storm of packets. In some such attacks, the attacker provides a false return (source) address, and the server sends the amplified stream at the

third party. In some cases the attacker provides a valid return address, but the excessive response by the server can provide DOS the attacker's ISP, or some link in the path to the attacker.

Buffer Bloat

The state of a connection where some buffer (a.k.a., queue) along the route from a source to a destination has grown large. This can typically appear at a bandwidth bottleneck having a higher ingress than egress rate. The result is seemingly unnecessary latency, induced by delay traversing the queue. The observation may be made that no matter how small or big (bloated) the buffer is, the bandwidth out the egress link is still fixed and bounded. Traditional TCP congestion control window will drive such a buffer at a bottleneck to its maximum size (inducing packet discards), and then back off, and again move toward this full (bloated) buffer state.

Connection

A bidirectional UDP connection over which data is transmitted. The side of the connection that first initiates the connection will create a Globally Unique ID (GUID) that will be effectively present in all UDP packets that are part of said connection. A connection is associated with a 4-tuple of (source IP/port, destinitation IP/port), just as it would be for TCP. Due to NAT in firewalls, the two ends of the connection may see slightly different 4-tuples, and commonly the source port and source IP may vary over time, but the GUID is completely invariant over the lifetime of the connection.

FEC

Forward Error Correction. The use of an error correcting code, which provides redundant information that is speculatively transmitted, to facilitate recovery of potentially lost packet data without a retransmission of the data. The simplest example of such a mechanism is a redundant transmission of the payload data in a packet, speculatively sent in a second packet. In that example, if the first packet is lost in transit, then the payload may be recovered from the redundant packet. A second example is to calculate the exclusive-OR sum of the payloads in a specific set of packets, and to speculatively transmit that sum in a redundancy (FEC) packet. In that example, if any single packet is lost, its contents may be recovered from the remaining packets, plus the FEC packet.

Frame

A portion of the encrypted payload inside a single UDP packet. Each UDP packet that has an encrypted payload contains one or more Frames. For example, a stream frame

effectively contains a byte-range of a stream that fits inside an encrypted payload. As a second example, an ACK frame contains acknowledgement related information, and fits inside an encrypted payload.

GUID

“Globally Unique ID.” A pseudo randomly generated 64 bit nonce, generated client side, that is expected to be unique from the perspective of the server, for the given (server) destination port. This should allow for about 2^32 distinct connections to be activated at the same time, at a given server port, based on client side (randomized) selection, without significant chance of a collision.

Packet

A single UDP packet, as sent over the underlying connection associated with a GUID.

Stream

One of potentially many data transmission channels for conveying data across a connection. A Stream is bidirectional. If the Stream is first created by the client (connection initiator), then it will have an odd-numbered stream ID. If the stream is created by the server (connection respondent), then it will have an even-numbered stream ID. The data in a Stream is automatically broken into Frames, and then re-assembled at the receiving end.

ACKNOWLEDGEMENTS

The ideas and descriptions listed here were gathered from many people, across many discussions and email threads. The contributors include, but are not limited to: Mike Belshe, Roberto Peon, Wan Teh Chang, Adam Langley, Will Chan, Yuchung Cheng, Nandita Dukkipati, Matt Mathis, Chris Bentzel, Eric Roman, Raman Tenneti, Ryan Hamilton, Alyssa Wilk, Michael Nowlan, Ian Fette, Patrik Westin, Barath Raghavan, Ryan Sleevi, Ian Swett, Vint Cerf, Matthew Dempsky, and Stuart Cheshire.

Those folks have responsibility for much of the cleverness expressed here, but are not to blame for any of the blatant misstatements and confusion provided. Some contributors may not even have reviewed this document in its entirety.

CHANGE NOTES

It is hard to track what has changed in this document. As a result, when I make a change (of substance, not just typos and minimal clarification) I'll try to add a note here.

- 6/24/2014: Revised and released as a publicly available document (via Google Docs) sponsored by the mozilla.org domain.
- 7/16/2012: Changed "NAT-PMP (NAT Port Mapping Protocol)" references to "Port Control Protocol (RFC 6887)"