2024-08-10 solana-quic-client fragmentation

Summary

Firedancer found a design flaw in Agave's QUIC client that forces additional complexity in the Firedancer QUIC server. We sent some patches to Anza to fix this situation.

Background

Our current fd_quic development strategy is to aggressively cut down on code complexity. This requires some sacrifices, mostly in the form of non-optional features and use-cases that generic QUIC clients would need but Solana peers do not. We did the same for fd_tls and it resulted in robust and statically verifiable code.

One such opportunity for simplification are QUIC streams. To recap, QUIC streams are comparable to TCP connections. They have unbounded size, receive windows, make no guarantees on fragment alignment, and generally can't deliver data out-of-order to clients.

fd_quic currently implements all the complexity associated with streams:

- receive windows for large transfers (MAX_STREAM_DATA frames)
- bidirectional streams
- reorder buffers (in case fragments from different streams arrive interleaved)
  - under the hood
- event queues and flags
- timeouts

With a well-designed client, none of that complexity would be needed. Solana peers use lots of small streams (MTU-ish <=1232 bytes) where each stream should fit in one or two QUIC packets. In particular, reorder buffers are redundant in the absence of fragmentation or with in-order fragmentation.

For example, this is a good traffic pattern:

```
Packet 1: Stream 1 (new=1 fin=1) # not fragmented
Packet 2: Stream 2 (new=1 fin=1) # not fragmented
Packet 2: Stream 3 (new=1 fin=0) # fragmented
Packet 3: Stream 3 (new=0 fin=1) # fragmented
Packet 3: Stream 4 (new=1 fin=1) # not fragmented
```

Whereas this pattern requires a more complex QUIC implementation:

```
Packet 1: Stream 1 (new=1 fin=0) # fragmented
Packet 1: Stream 2 (new=1 fin=0) # fragmented
Packet 2: Stream 1 (new=0 fin=0) # fragmented, out of order
Packet 3: Stream 2 (new=0 fin=1) # fragmented, out of order
```

The second traffic pattern is bad because it requires the peer to allocate buffers for streams 1 and 2 that live for multiple packets. The peer also wouldn't know how bad the fragmentation gets so it has to assume the worst case: Every transaction that the remote is allowed to send might be made up of out-of-order fragments. (Note on send allowance: QUIC has a connection-wide limit on the number of new streams as well as per-stream content size)

As a result, handling a bad client requires a lot more memory. The memory pressure in turn limits maximum throughput.

solana-quic-client issues

Unfortunately, solana-quic-client produces an odd traffic pattern in even under optimal network conditions.

We wrote a simple test that connects the Agave QUIC client implementation (solana-quic-client) to an fd_quic server. The QUIC client is configured to send transactions as fast as possible using a single thread and a single connection. A uniform size distribution is used for packets. The peers are connected via localhost. Source: firedancer: Add spam-server mode to quic-compat CLOSED

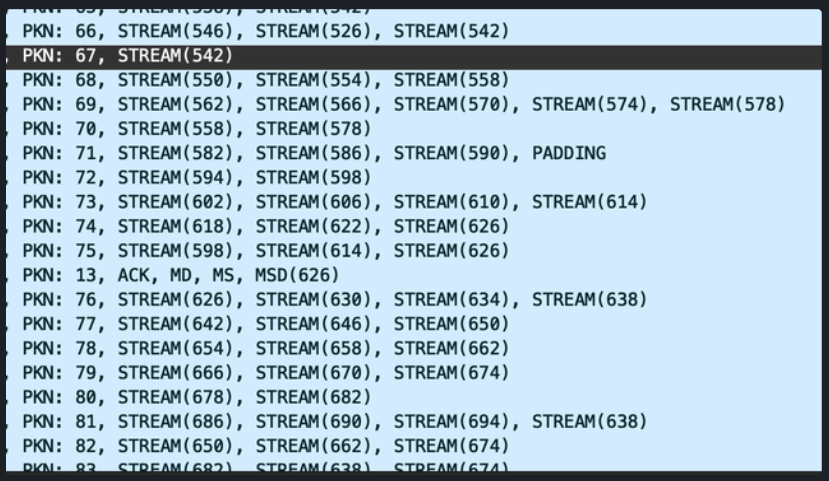

Figure 1: Upstream solana-quic-client traffic pattern

Figure 1 shows the resulting traffic pattern. Note how many stream fragments are out of order: 526, 558, 578, 598, 614, 626, 638, 650, 662, …

Reverse Engineering

Note that the client here turned an ordered sequence of messages into a partially reordered sequence of fragments. This doesn't just happen accidentally and requires some effort. Possible culprits include the Agave code itself and quinn, the QUIC library it uses.

Tracing from top down:

```
// Our custom test app
let mut batch = Vec::<Vec<u8>>::with_capacity(1024);
const BUF: [u8; 1232] = [0u8; 1232];
loop {
    let cnt: usize = rng.gen_range(1..batch.capacity());
    batch.clear();
    for _ in 0..cnt {
        batch.push(BUF[0..rng.gen_range(1..BUF.len())].to_vec());
    }
    if let Err(err) = conn.send_data_batch(&batch) {
        eprintln!("{:?}", err);
    }
}
```

In typical Agave fashion, the function is thunked a few times:

```
dispatch!(fn send_data_batch(&self, buffers: &[Vec<u8>]) -> TransportResult<()>);
// calls
fn send_data_batch(&self, buffers: &[Vec<u8>]) -> TransportResult<()> { ... }
// calls
async fn send_data_batch(&self, buffers: &[Vec<u8>]) -> TransportResult<()> { ... }
// calls
pub async fn send_batch<T>(...) -> { ... }
```

The problem becomes apparent in `send_batch` (as of Agave v1.18.22 and v2.0.4). Comments and metrics were removed for brevity:

```rust
async fn _send_buffer_using_conn(
    data: &[u8],
    connection: &Connection,
) -> Result<(), QuicError> {
    let mut send_stream = connection.open_uni().await?;
    send_stream.write_all(data).await?;
    send_stream.finish().await?;
    Ok(())
}

pub async fn send_batch<T>(&self, buffers: &[T]) -> Result<(), ClientErrorKind>
    where T: AsRef<[u8]>,
{
    if buffers.is_empty() {
        return Ok(());
    }
    let connection = self
        ._send_buffer(buffers[0].as_ref())
        .await
        .map_err(Into::<ClientErrorKind>::into)?;
    let connection_ref: &Connection = &connection;
    let chunks = buffers[1..buffers.len()].iter().chunks(self.chunk_size);
    let futures: Vec<_> = chunks
        .into_iter()
        .map(|buffs| {
            join_all(
                buffs
                    .into_iter()
                    .map(|buf| Self::_send_buffer_using_conn(buf.as_ref(), connection_ref)),
            )
        })
        .collect();
    for f in futures {
        f.await
            .into_iter()
            .try_for_each(|res| res)
            .map_err(Into::<ClientErrorKind>::into)?;
    }
    Ok(())
}
```

In short, this code creates a “Future” for every transaction. (A Rust future is a background task that the user “polls” repeatedly until completion. Each poll operation is non-blocking)

The `_send_buffer_using_conn` future includes 3 steps (each one is a child “Future”):

1. `quinn::OpenUni`: Open a unidirectional stream (usually completed instantly, may block if client has run out of quota).
2. `quinn::WriteAll`: Send a sequence of bytes over the stream (usually completed instantly, may block if stream- or connection-wide quota limits are hit)
3. `quinn::Finish`: Wait for the peer to acknowledge all packets that contained stream data (always incurs network delay)

Waiting for the server to ACK every stream is obviously too slow so the Agave code attempts to transmit multiple streams “concurrently” in chunks. Of course, there is no true “concurrent transmission”. Outgoing packets will be sequenced to some order before being sent to the Ethernet channel. And herein lies the problem. WriteAll yields data fragments more than once before completing. The order in which `_send_buffer_using_conn` Futures are polled is unspecified.

Depending on the version of the “futures” and “quinn” libraries, the code might send every transaction in order (good), or take round robin turns sending small fragments (terrible).

Rewrite

A good first fix is to remove the ACK delay to allow for maximal pipelining. The QUIC library handles packet loss (retransmissions) under the hood anyways.

```
async fn _send_buffer_using_conn(
    data: &[u8],
    connection: &Connection,
) -> Result<(), QuicError> {
    let mut send_stream = connection.open_uni().await?;
    send_stream.write_all(data).await?;
    // send_stream.finish().await?;  <== remove this
    Ok(())
}
```

The `send_batch` function is simplified to the following:

```
pub async fn send_batch<T>(&self, buffers: &[T]) -> Result<(), ClientErrorKind>
    where T: AsRef<[u8]>,
{
    if buffers.is_empty() {
        return Ok(());
    }
    let connection = self
        ._send_buffer(buffers[0].as_ref())
        .await
        .map_err(Into::<ClientErrorKind>::into)?;
    for data in buffers[1..buffers.len()].iter() {
        Self::_send_buffer_using_conn(data.as_ref(), &connection).await?;
    }
    Ok(())
}
```

Figure 2 shows the resulting packet sequence.

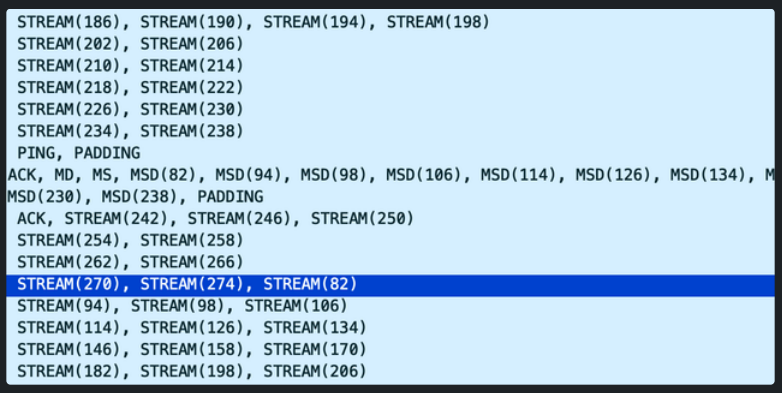

Figure 2: Rewrite Attempt

Even worse: We got round-robin fragmentation instead of sporadic reordering. Note that the server set a sufficient initial_max_stream_data so we can rule out potential blocking by stream window sizes.

quinn bug

This behavior is presumably due to a bug in quinn 0.10.2. One would expect the following code to produce an ordered fragment stream:

```
loop {
    let mut stream = conn.open_uni().await?;
```

```rust

    // await polls the WriteAll Future to completion.
    stream.write_all(...).await?;

    // All data was enqueued into QUIC packets ready to be sent, right?
    // Hint: No.
}
```

Unfortunately this is not the case. Stream fragments compete for priority somewhere in quinn's internals.

Increasing the stream's "priority" after the `write_all` operation fixes this behavior. Note that priority isn't a thing in QUIC. It appears to be some implementation-defined setting that controls transmit fragment ordering.

```rust
loop {
    let mut stream = conn.open_uni().await?;
    stream.write_all(...).await?;
    // Finish sending this stream before sending any new stream
    _ = send_stream.set_priority(1000);
}
```

At last, we get well ordered fragment stream.

```
PKN: 8, ACK, MD, MS
PKN: 95, STREAM(778), STREAM(782)
PKN: 96, STREAM(782), STREAM(786), STREAM(790), STREAM(794)
PKN: 97, STREAM(794), STREAM(798), STREAM(802), STREAM(806), STREAM(810)
PKN: 98, STREAM(810), STREAM(814), STREAM(818)
PKN: 9, ACK, MD, MS
PKN: 99, ACK, STREAM(822), STREAM(826), STREAM(830), STREAM(834)
PKN: 100, STREAM(834), STREAM(838), STREAM(842), STREAM(846)
PKN: 101, STREAM(846), STREAM(850), STREAM(854)
PKN: 102, STREAM(854), STREAM(858), STREAM(862), STREAM(866), STREAM(870)
PKN: 103, STREAM(870), STREAM(874), STREAM(878), STREAM(882)
PKN: 10, ACK, MD, MS
PKN: 104, ACK, STREAM(886), STREAM(890)
PKN: 105, STREAM(890), STREAM(894)
PKN: 11, ACK, MD, MS
PKN: 106, STREAM(898), STREAM(902), STREAM(906)
PKN: 107, STREAM(906), STREAM(910)
PKN: 12, ACK, MD, MS
PKN: 108, ACK, STREAM(914), STREAM(918), STREAM(922), STREAM(926)
PKN: 109, STREAM(926), STREAM(930), STREAM(934), STREAM(938)
PKN: 110, STREAM(938), STREAM(942), STREAM(946), STREAM(950), STREAM(954), STREAM(958
PKN: 111, STREAM(958), STREAM(962)
PKN: 112, STREAM(962), STREAM(966), STREAM(970), STREAM(974), STREAM(978)
PKN: 113, STREAM(978), STREAM(982), STREAM(986)
PKN: 114, STREAM(986), STREAM(990), STREAM(994)
PKN: 115, STREAM(994), STREAM(998), STREAM(1002), STREAM(1006)
PKN: 116, STREAM(1006), STREAM(1010), STREAM(1014), STREAM(1018)
PKN: 117, STREAM(1018), STREAM(1022), STREAM(1026)
```

Figure 3: Fixed solana-quic-client with workaround

Firedancer submitted the patch to Agave here. It weights in at a net 86 line code footprint reduction.

solana: Fix broken parallelism in quic-client OPEN

Created by ripatel-fd • Updated yesterday

Fixes excessive fragmentation by TPU clients leading to a large number of streams per conn in 'sendi

Github

fd_quic server improvements

Still need to write this part.

Fragmentation

There is no need to fragment transactions at all, at least at the moment. As of Solana v2.0 the largest permitted transaction size is 1232 bytes. This is comfortably below the QUIC stream frame payload MTU over IPv4. Unfortunately such fragmentation is required for IPv6 due to botched MTUs and ICMPv6 blocking across deployments in the wild.

Removing fragmentation will increase the packet rate though. Further research is required to determine whether Agave's per-packet overheads are low enough.

QUIC protocol limitations

All of the above is a consequence of coarse protocol limits. The MAX_STREAMS limit in QUIC controls roughly "*how many new streams may be started spontaneously*". This also implies "*how many streams may be in send state concurrently*" (all of them) but this should be a separate limit.