The Quake3 Networking Model

This article is a reprint/updated version of an e-mail I sent out to the mud-dev mailing list a few years ago, describing the Q3 networking model as described to me by John Carmack. I did this partly out of my own curiousity (since I was firmly entrenched in the graphics side of things) and partly out of a desire to propagate information on the 100% unreliable networking model in Q3 which I felt (and still feel) was fairly groundbreaking due to its simplicity and ease of understanding.

The First Attempt (QTEST/Quake2)

Carmack's first real networking implementation, back in 1995, used TCP for QuakeTest (QTEST). This was fine for LAN play, because the large packets (8K) wouldn't fragment on a LAN (by "fragment", I mean to the point where disassembly and reassembly induced significant overhead), but didn't work so well over the Internet due to fragmentation (where dis/reassembly and lost packets often resulted in very expensive resends). His next iteration involved using UDP with both reliable and unreliable data, pretty much what many would consider a standard networking architecture. However standard mixed reliabled/unreliable implementations tend to generate very hard to find bugs, e.g. sequencing errors where guaranteed messages referenced entities altered through unreliable messages.

Quake3

The final iteration (Quake3), which was the first time he really felt he "got it right" (whereas Carmack always felt a little uneasy with previous implementations' fragility), used a radically different approach. With Quake3 he dropped the notion of a reliable packet altogether, replacing the previous network packet structure with a single packet type -- the client's necessary game state. The server sends sequenced game state updates that are delta compressed from the last acknowledged game state the client received. This "just works". Dropped packets are handled implicitly, and specific commands are never acknowledged independently -- last known state acks are piggy backed on regular incoming/outgoing traffic.

The client's receive logic boils down to:

```
if ( newState.sequence < lastState.sequence )
{
   //discard packet
}
else if ( newState.sequence > lastState.sequence )
{
   lastState = deltaUncompress( lastState, newState );

   ackServer( lastState.sequence );
}
```

The client then extrapolates movement and whatever other information it needs based on the last game state it received.

It's even simpler on the server:

```
deltaCompressState( client.lastAckState, newState, &compressedState );

sendToClient( client, compressedState );
```

The server never sits around waiting for an acknowledgement. As a result, latencies are much lower than if you have code that sits there twiddling its thumbs waiting for a synchronous ACK.

There are two downsides to this implementation. The big one is that it soaks more bandwidth since the server is constantly pumping new state to the client instead of just sending new state when it doesn't get an ACK. Also, the amount of data sent grows with the number of dropped packets since the delta grows as a result.

The other downside is that the server must buffer a lot of data, specifically of the last acked state to the client (so it can delta compress) along with a list of all the previous states it has sent to the client (back to the last acked one). This is necessary so it can rebase its delta comparisons on any of the game states that the client has acked.

For example, let's say the client last acked sequence 14. The server is sending out new game states with incrementing sequences. It may be up to sequence 20 when it gets an ack from the client that it has received sequence 17. The server has to have the state that was sent as part of sequence 17 in order to rebase the delta compression, so if game states are large enough or the buffers are long enough, this can grow fairly high (to the tune of several hundred K per client).

All reliable data that another node needs is sent repeatedly until the sender receives an update for most-recent-ack (indicating that the packet has been received). For example, if a player sends a chat message (reliable) with update 6, he will continually send that chat message on subsequent state updates until he receives notification from the server that it has received an update >= 6. Brute force, but it works.

Port Allocation

A note about using multiple ports -- there is one significant advantage to using multiple ports, and that's that the OS buffers incoming data per port. So if you have a chance of a port buffer overflow, then multiple ports may not be a bad idea. If a port overflow is highly unlikely to occur, then multiple ports probably just complicate matters.

Speaking of port buffer overflows, John doesn't think this is a problem. People that spin a second thread just to block on incoming ports are probably just doing extra work that doesn't need to be done. Effectively a thread that pumps the data ports is duplicating the exact same work the network stack is doing in the OS. Instead, John just pumps the network stack at the top of the event loop. Goes to show that brute force sometimes "just works". On a listen server (i.e. you're hosting a server on your client), where frame times can be several dozen milliseconds, there is a chance that you'll get a buffer overrun. In that case there are some options in Q3 to minimize buffer sizes, etc. to reduce the likelihood of a buffer overrun.

One nice thing about Q3's networking architecture, however, is that even in the case of a buffer overrun it just keeps working. There is no effective difference to this system between a dropped packet and a buffer overrun; as a matter of fact, this may actually be masking some real buffer overruns, but so far Carmack's pretty sure that he's not even come close to hitting an overrun situation. Of course, the dynamics of a shooter are different than those of an MMOG (fewer clients, much higher bandwidth per client), but it's interesting nonetheless.

NAT

One interesting note on NAT: apparently some older routers will randomly switch the client's port when behind a NAT. This is very rare, but causes lost connections "randomly" for many users. It took forever to track this down, but once it was discovered he solved it by having a

client randomly generate a unique client-id (16-bits) at connection time that is appended to every incoming packet. That way multiple clients with the same IP can work just fine, even if their ports get re-assigned mid-session. Humorously enough, he never bothered handling the (so rare as to be insignificant) case where two clients behind the same firewall randomly generate the exact same ID (in the event this occurred one of the clients would be dropped due to badly out of sync state).

Compression, Encryption, and Packets

Aside from application level delta compression, Q3 also does per-packet Huffman compression. Works great, simple to implement, and the upside to something more aggressive is very low.

He has done some half-hearted work on encryption, but basically app/stream-level encryption is pointless because of the sophistication of hackers. In the future he'll probably rely on higher level inspection (a la Punkbusters for Counter-Strike) instead of cute bit-twiddling.

Finally, I asked about packet size. He thinks the notion of a 512-byte optimal MTU is pretty much at least 5 years out of date. He sees almost no problems with a max packet size of 1400 bytes, and fragmentation just isn't an issue. During the course of typical activity, the actual payload size is often smaller than the UDP header size (!), and even during hectic activity the packet size is on the order of several hundred bytes. The large packets (up to 1.4K) are typically when a large amount of game state must be sent at once, such as at connection time when there is no state to delta against and when there is a lot of startup initialization going on.

Summary

The Q3 networking model obviates the need to even have a discussion about UDP vs. TCP, unreliable vs. reliable, and out-of-order vs. in-order. It's all unreliable UDP delta compressed against the last known state.

A very elegant and effective architecture, albeit one that is ideally suited to a subset of game types.